J Autom Reasoning (2008) 41:219–249
DOI 10.1007/s10817-008-9108-3

An Extensible Encoding of Object-oriented Data Models in HOL

with an Application to IMP++

Achim D. Brucker · Burkhart Wolff

Received: 14 November 2008 / Accepted: 14 November 2008 / Published online: 14 December 2008

Abstract We present an extensible encoding of object-oriented data models into higher-order logic (HOL). Our encoding is supported by a datatype package that leverages the use of the shallow embedding technique to object-oriented specification and programming languages. The package incrementally compiles an object-oriented data model, i. e., a class model, to a theory containing object-universes, constructors, accessor functions, coercions (casts) between static types (and providing a foundation for the notion of dynamic types), characteristic sets, and co-inductive class invariants. The package is conservative, i. e., all properties are derived entirely from constant definitions, including the constraints over object structures. As an application, we use the package for an object-oriented core-language called IMP++, for which we formally prove the correctness of a Hoare logic with respect to a denotational semantics.

Keywords object-oriented data models · HOL · theorem proving · verification

1 Introduction

While object-oriented programming is a widely accepted programming paradigm, theorem proving over object-oriented programs or object-oriented specifications is

A.D. Brucker (✉)
SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
e-mail: achim.brucker@sap.com
URL: `http://www.brucker.ch/`

B. Wolff
Université Paris-Sud, Parc Club, 4, Rue Jaques Monod, 91893 Orsay Cedex, France
e-mail: wolff@lri.fr
URL: `http://www.lri.fr/~wolff`

far from being a mature technology. Classes, inheritance, subtyping, objects and references are deeply intertwined and complex concepts that are quite remote from the platonic world of first-order logic (FOL) or higher-order logic (HOL). For this reason, there is a tangible conceptual gap between the verification of functional programs and object-oriented programs.

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

1. *verification condition generators* reducing a Hoare-style proof into a proof in a standard logic, and
2. *deep embeddings* into a meta-logic.

Verification condition generators, for example, are Boogie for Spec# [5,21], Krakatoa [22] and several similar tools based on the Java Modeling Language (JML) [20]. 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: besides the correctness problem of the compilation, there is the problem that the operations of the target machine (i. e., the memory model) must be axiomatized; for Spec#, this easily results in several hundreds of axioms. Even if one believes that these axiomatizations adequately describe the target machine, the question of consistency of these axiomatizations is highly non-trivial [9].

Among the tools and formalizations based on deep embeddings, there is a sizable body of literature on formal models of Java-like languages (e. g., [15,16,28,31]). 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 the 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 over applications and not over language tweaks, this advantage turns into a major obstacle for efficient deduction. Thus, while various proofs for type-safety, soundness of Hoare calculi and even soundness of verification condition generators are formally proven, none of the mentioned deep embeddings has been used for substantial proof work in applications.

In contrast, the *shallow embedding* technique has been used for semantic representations such as HOL itself (in Isabelle/Pure), for HOLCF (in Isabelle/HOL) used for reasoning over Haskell-like programs [25] or for HOL-Z [11,6].

The essence of an effective shallow embedding is to find an injective mapping of the pair of an object-language expression `E` and its type `T` to a pair $E :: T$ of the meta-language HOL. "Injective mapping" means, that well-typedness is preserved in both ways. Thus, type-related side-conditions in derived object-language calculi can be left implicit. Since such implicit side-conditions are "implemented" by a built-in mechanism of the meta-logic, they can be checked orders of magnitude faster compared to an explicit treatment involving tactic proof.

At first sight, it seems impossible to apply the injective shallow embedding technique to object-oriented languages: Their characteristic features like subtyping and inheritance are not present in the typed λ-calculi underlying HOL systems. However, an injective mapping in our sense does not mean a simple one-to-one conversion; rather, the translation might use a pre-processor, making, for example, implicit casts between subtypes and supertypes explicit. Still, we need a formal data model that gives first-class support such semantic properties, like type casting without losing information.

Beyond the semantical requirements, there is an important technical one: object-oriented data models must be extensible, i. e., it must be possible to add a class to an existing class model (or class system) without reproving everything. The problem becomes apparent when considering the underlying *state* of an object-oriented program. This state consists of *objects*, i. e., abstract representations of pieces of memory, that may be linked via references (object identifiers, oid) to each other. Objects are tuples of class attributes, i. e., elementary values like Integers or Strings or references to other objects. The type of these tuples is interpreted as the type of the class they are belonging to. States are maps of type oid $\Rightarrow \mathscr{U}$ relating references to objects living in a universe $\mathscr{U}$ of all objects.

Instead of constructing such a universe globally for all data models (which is either single-typed and therefore not an injective type representation, or "too large" for the type system of HOL), one could think of generating an object universe only for each given class model. Ignoring subtyping and inheritance for a moment, this would result in a universe $\mathscr{U}^0 = A + B$ for some class model with the classes A and B. Unfortunately, such a construction is not extensible: If we add a new class to an existing class model, say D, then the construction $\mathscr{U}^1 = A + B + D$ results in a type *different* from $\mathscr{U}^0$, making their object structures logically incomparable. Properties, that have been proven over $\mathscr{U}^0$ will not hold over $\mathscr{U}^1$. Thus, such a naive approach rules out an incremental construction of class models, which makes it unfeasible in practice. This holds in particular for an interactive theorem proving approach.

As contributions of this paper, we present a novel universe construction which represents subtyping within parametric polymorphism in an injective, type-safe manner *and* which is extensible. This construction is implemented in a datatype package for Isabelle/HOL, i. e., a compiler that generates for each class model and its extensions conservative definitions. This includes the definition of constructors and accessors, casts between types, type tests, and characteristic sets of objects. We apply this specification infrastructure by integrating it into the assertions of a Hoare calculus for a small object-oriented language called IMP++.

The plan of the paper is outlined as follows: In Section 2 we introduce our meta-language higher-order logic (HOL) and a syntactic interface for accessors, casts, tests, etc, in the form of the assertion language COOL. In the following sections, we present our type-safe encoding of COOL by several consecutive levels: In Section 3, we present level 0 which provides a typed representation of objects (i. e., pieces of memory) in a store. In Section 4, we present level 1 of our COOL encoding which provides a type-safe implementation of its accessors, casts, tests, ... In Section 5 we give a more refined definition of COOL, called level 2 encoding, which supports class invariants. We show in Section 6 in which ways our constructions support modular proof methodologies. Section 7 contains the application part of the paper: we provide a denotational semantics for IMP++ and derive a calculus for Hoare-style program verification. Finally, in Section 8 we draw conclusions and discuss related work.

2 Preliminaries

2.1 Isabelle/HOL

Isabelle [29] is a generic, LCF-style theorem prover implemented in the functional programming language SML. For our object-oriented datatype package, we use the possibility of using Isabelle for building SML programs performing symbolic computations over formulae in a logically safe way. Isabelle/HOL supports conservativity checks of definitions, datatypes, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableau provers.

Higher-order logic (HOL) [4] is a classical logic with equality enriched by total polymorphic higher-order functions. The type constructor for the function space is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\cdots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\cdots)$ are also written as $[\tau_1,\ldots,\tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha,\alpha] \Rightarrow \text{bool}$, where bool is the fundamental logical type.

The type discipline rules out paradoxes such as Russel's paradox in untyped set theory. Sets of type $\alpha\,\text{set}$ can be defined isomorphic to functions of type $\alpha \Rightarrow \text{bool}$; the element-of-relation $\_ \in \_$ has the type $[\alpha, \alpha\,\text{set}] \Rightarrow \text{bool}$ and corresponds basically to the application; in contrast, the set comprehension $\{\_|\_\}$ has type $[\alpha\,\text{set}, \alpha \Rightarrow \text{bool}] \Rightarrow \alpha\,\text{set}$ and corresponds to the λ-abstraction.

The HOL type constructor $\tau_\perp$ assigns to each type τ a type *lifted* by $\perp$; for lifted types a test for definedness is available via $\text{def}x \equiv (x \neq \perp)$. The function $\lfloor\_\rfloor : \alpha \Rightarrow \alpha_\perp$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\perp \Rightarrow \alpha$ its inverse for defined values. Based on these definitions, partial functions, i. e., $\alpha \rightharpoonup \beta$, can be represented as total functions of type $\alpha \Rightarrow \beta_\perp$. We define *dom* f, called the *domain* of a partial function f, by the set of all arguments of f that do not yield $\perp$.

2.2 COOL—A Core Object-oriented Assertion Language

In this section, we introduce a core object-oriented assertion language (COOL). We start by presenting the concrete syntax for COOL at a glance and thus make the domain of our logical representation more precise. The basic lexical entities of COOL are:

- C, a set of *class names*,
- A, a set of *attributes*,
- V, a set of *variable symbols*, and
- X, a set of *program variable symbols*.

A class is a component consisting of an abstract name and a set of attributes. Thus, technically, the type of a class is based on its abstract name and the types of its attributes. On the level of COOL, we identify the type of a class by its name. Moreover, COOL supports the usual basic datatypes, e. g., Integer, and provides a range of collection types. In more detail, the *types* of COOL are inductively defined as follows:

$$\begin{aligned} T := C \mid \texttt{Boolean} \mid \texttt{Integer} \mid \texttt{Real} \mid \texttt{String} \mid T\,\texttt{Sequence} \\ \mid T\,\texttt{Set} \mid T\,\texttt{Bag} \mid T \succ T\,. \end{aligned} \tag{1}$$

In Section 3 and Section 4 we describe how these types can be mapped injectively to HOL types.

As abstract syntax, a class model, also called class system, is a (finite) partial map that assigns to a class identifier a (finite) map, that associates to each attribute name its type:

$$M := C \rightharpoonup (A \rightharpoonup T)\,. \tag{2}$$

For simplicity, we assume that attribute names are unique throughout this paper; this means that the domains of two elements in the range of M are always pairwise disjoint. Our formalization in Isabelle/HOL, and thus our datatype package, require only that attribute names are unique within a class.

We assume an irreflexive partial order $\_ < \_$ on class types called *class hierarchy* and for $c_i < c_j$, we will say that the class c_i is a *subclass* of class c_j. The following inductive set of expressions is called *path expressions*:

$$P := V \mid P[C] \mid P.A\,. \tag{3}$$

Path expressions are build from program variables, castings to a class type, and accessor functions to an attribute of an object. The expressions of COOL are:

$$\begin{aligned} E := {} & P \mid \partial P \mid P.\mathtt{isType}\;(T) \mid P.\mathtt{isKind}\;(T) \mid \\ & V \mid E = E \mid \neg E \mid E \rightarrow E \mid \cdots \mid \\ & \forall V.E \mid \exists V.E \mid \\ & E + E \mid E * E \mid \cdots \mid E \cup E \mid \cdots \mid \{V.E\} \mid \cdots \end{aligned} \tag{4}$$

While the latter lines sketch a conventional expression language, the first line contains key elements of object-orientation: the result of a path expression may denote (sets of, sequences of) objects, paths can be tested if they are *defined* (∂P) within a store, the *dynamic type* ($P.\mathtt{isType}\;(T)$) and the *kind* ($P.\mathtt{isKind}\;(T)$) of an object may be tested. Recall that casts can change the static type of an object, while the dynamic type is just the type of an object at creation time; the *kind* of an object is defined as the dynamic type and all its subtypes. These concepts can be found in many object-oriented languages, e. g., Java, C#, and UML/OCL.

We refrain from a presentation of the (obvious) type inference rules for COOL. An assertion over a state σ in COOL is denoted as $\sigma \models E$. Semantically, an assertion is a set of states σ that satisfy E (which must be of type `Boolean`).

We present our framework for encoding object-oriented data structures and assertion languages together with a small example (see Figure 1): we assume a class `Node` with an attribute `i` of type `Integer` and the attributes `left` and `right`, both of type `Node`. Further, we assume a subclass `CNode` of `Node` with an attribute `color` of type `Boolean`. Thus, we have `CNode` $<$ `Node` in the subtype ordering, and $\{\mathtt{CNode} \mapsto \{\mathtt{color} \mapsto \mathtt{Boolean}\}, \mathtt{Node} \mapsto \{\mathtt{left} \mapsto \mathtt{Node}, \ldots\}\}$. Note, however, that the COOL assertions are based on a two-valued semantics (similar to Spec# or JML), whereas OCL [1] (a constraint language for UML [2]) is based on a three-valued logic, i. e., the type `Boolean` ranges over `true`, `false`, and `OclUndefined`.

While our datatype package was developed within the HOL-OCL project [13,12], it can be used for an arbitrary assertion language with the techniques presented here. When a specification is loaded, all definitions presented in Section 3, Section 4, and Section 5, are automatically generated and all theorems automatically proven by our datatype package. Besides the concrete syntax presented throughout this paper, our

```
class Node
  attributes
    i     : Integer
    left  : Node
    right : Node
  constraints
    inv positive: self.i > 5
end

class CNode < Node
  attributes
    color : Boolean
  constraints
    inv flip:
          ∂ self.left[CNode] ⟶ self.color = ¬self.left[CNode].color
        ∧ ∂ self.right[CNode] ⟶ self.color = ¬self.right[CNode].color
    inv flip_strong:
          ∂ self.left[CNode] ∧ self.color = ¬self.left[CNode].color
        ∧ ∂ self.right[CNode] ∧ self.color = ¬self.right[CNode].color
end
```

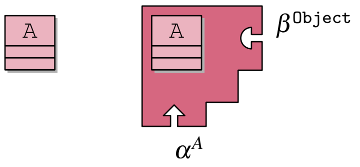
Figure 1 A simple class model describing directed graphs: we model a class `Node` with an attribute `i` of type `Integer` and two attributes `left` and `right` for storing adjacent objects of type `Node`. Moreover, we model a subclass `CNode` of `Node` that introduces an attribute `color` of type `Boolean`. Both classes are described more precisely by invariants, e. g., for any instance of class `Node` we require that the value of `i` is larger than five. For the class `CNode`, we require two invariants, a weaker and a stronger one. Both require that the `color` attribute flips its value while traversing a path through the object graph, the second one also requires additionally that left and right nodes exist.

datatype package is also able to load UML class models (together with an OCL specification) in a standardized exchange format (XMI) used by most CASE tools.

3 Level 0: Typed Object Universes

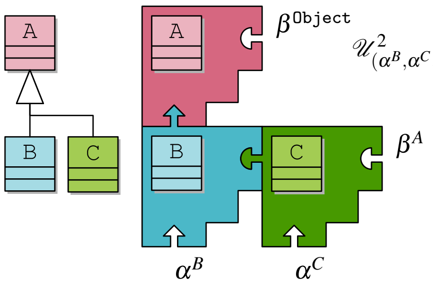
A key objective of our approach is to type objects unambiguously. Although the objects may contain untyped object identifiers (oid's), the operations of COOL will be defined such that they always work on objects, not on object identifiers.

As mentioned earlier, a consequence of typed objects is the necessity of object universes to define the store model in HOL. In this section, we introduce our families $\mathscr{U}^i$ of object universes enabling extensibility. Here, we use the superscript to denote the temporal development, i. e., $\mathscr{U}^{i+1}$ is originated by extending $\mathscr{U}^i$. Each $\mathscr{U}^i$ comprises all *value types* and an extensible *class type representation* induced by the class hierarchy. To each class, a *class type* is associated which represents the set of *object instances* or *objects*. In other words, a type is constructed for all objects ("pieces of memory") belonging to a class. The extensibility of a universe type is reflected by "holes" (polymorphic variables), that can be filled when "adding" extensions to a class system. Our construction ensures that $\mathscr{U}^{i+1}$ is just a type instance of $\mathscr{U}^i$ (where $\mathscr{U}^{i+1}$ is constructed by adding new classes to $\mathscr{U}^i$). Thus, properties proven over object systems "living" in $\mathscr{U}^i$ remain valid in $\mathscr{U}^{i+1}$, see Figure 2 for an illustration of the main ideas of the construction we present in the following.



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

(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.



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

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

Figure 2 Assume we have a model consisting only of one class A which "lives" in the universe $\mathscr{U}^1_{(\alpha^A,\beta^{\texttt{Object}})}$ 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 $\mathscr{U}^2_{(\alpha^B,\alpha^C,\beta^A,\beta^{\texttt{Object}})}$ for the final class hierarchy. In particular, the universe $\mathscr{U}^2_{(\alpha^B,\alpha^C,\beta^A,\beta^{\texttt{Object}})}$ is a type instance of $\mathscr{U}^1_{(\alpha^A,\beta^{\texttt{Object}})}$. Thus, properties that have been proven over the initial universe $\mathscr{U}^1_{(\alpha^A,\beta^{\texttt{Object}})}$ are still valid over the extended universe $\mathscr{U}^2_{(\alpha^B,\alpha^C,\beta^A,\beta^{\texttt{Object}})}$.

3.1 A Typed Object Store

In the following we define several type sets which all are subsets of the types of the HOL type system. These sets, although denoted in usual set-notation, are a meta-theoretic construct, i. e., they cannot be formalized in HOL.

First, we introduce types for attributes. Since we aim towards a conservative formalization of our object store, we cannot model the set T of COOL types directly we have to support for mutually recursive data structures. Thus, we introduce a special type oid for object-identifiers, i. e., we assume that each object (instance of a class) can be uniquely identified by its identifier (which is of type oid) and we formally define the set $\mathfrak{A}$ of attribute types and the set $\mathfrak{C}$ of class types. Conceptually, the union of these two type sets provides a first representation of the set T of COOL types. Formally, we define for the class attributes the set of attribute types as follows:

Definition 1 (Attribute Types) The set of *attribute types* $\mathfrak{A}$ is defined inductively as follows:

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

Attributes with non-value types, e. g., the attribute `left` of class `Node`, are encoded using the type oid. Of course, this representation cannot guarantee the type-safety for attributes with non-value types. Therefore, these object identifiers (i. e., references) will be resolved by accessor functions, on level 1 (see Section 4 for details) of our encoding, like $A.\mathrm{left}^{(1)}$ (the superscript denotes that this is an attribute accessor on level 1) for a given state; an access failure will be reported by $\perp$.

The main idea of the following encoding of class types is as follows: we represent a class by a tuple, which is built by pairing the attribute types of the class. Moreover, we extend this type by an abstract datatype for each class. This abstract datatype, the *tag type*, ensures that there is a bijection between the class types and their representation as tuple.

Definition 2 (Tag Types) For each class C a *tag type* $\mathrm{C_t} \in \mathfrak{T}$ is associated. The set $\mathfrak{T}$ is called the set of tag types.

Tag types are one of the reasons why we can build a strongly typed universe (with respect to the object-oriented type system), e. g., for class `Node` we introduce an abstract datatype $\mathrm{Node_t}$ with the only element $\mathrm{Node_{key}}$. Further, for each class we introduce a base class type:

Definition 3 (Base Class Types) The set of *base class types* $\mathfrak{B}$ is defined as follows:

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

Thus, the base type of an object of class `Node` is $\mathrm{Node_t} \times$ `Integer` $\times\, \mathrm{oid} \times \mathrm{oid}$ and of class `CNode` is $\mathrm{CNode_t} \times$ `Boolean`. Conceptually, the *base class type* represents the mapping of COOL class names C (represented by the tag type) to the attributes A (directly represented by their types).

Without loss of generality, we assume in our object model a common supertype of all objects. For example, for Java this is `Object`, while for OCL, this is `OclAny`.

Definition 4 (Object) Let $\mathrm{Object_t} \in \mathfrak{T}$ be the tag of the common supertype `Object` and oid the type of the object identifiers. We define α `Object` $:= \big((\mathrm{Object_t} \times \mathrm{oid}) \times \alpha_\perp\big)$.

Object generator functions can be defined such that freshly generated object-identifiers to an object are also stored in the object itself; thus, the construction of reference types and of referential equality is fairly easy (see the discussion on state invariants in Section 6.4). Now we have all the foundations for defining the type of our family of universes formally:

Definition 5 (Universe Types) The set of all universe types $\mathfrak{U}$ is inductively defined by:

1. $\mathscr{U}^0_\alpha \in \mathfrak{U}$ is the initial universe type with one type variable (hole) α.
2. if $\mathscr{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)} \in \mathfrak{U}$, $n,m \in \mathbb{N}$, $i \in \{0,\dots,n\}$ and $c \in \mathfrak{B}$ then
$$\mathscr{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)}\Big[\alpha_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}$$

This definition covers the introduction of "direct object extensions" by instantiating α-variables.

3. if $\mathcal{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)} \in \mathfrak{U}$, $n,m \in \mathbb{N}$, $i \in \{0,\dots,m\}$, and $c \in \mathfrak{B}$ then
$\mathcal{U}_{(\alpha_0,\dots,\alpha_n,\beta_1,\dots,\beta_m)} \Big[\beta_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}$
This definition covers the introduction of "alternative object extensions" by instantiating β-variables.

The initial universe $\mathcal{U}^0_\alpha$ represents mainly the common supertype (i. e., `Object`, `OclAny`) of all classes, i. e., a simple definition would be $\mathcal{U}^0_\alpha = \alpha$ `Object`. However, we will need the ability to store *Values* $=$ `Real` $+$ `Integer` $+$ `Boolean` $+$ `String`. Therefore, we define the initial universe type by $\mathcal{U}^0_\alpha = \alpha$ `Object` $+$ *Values*. Extending the initial universe $\mathcal{U}^0_\alpha$ with the classes `Node` and `CNode` leads to the following universe type:

$$\mathcal{U}^1_{(\alpha_\mathrm{C},\beta_\mathrm{C},\beta_\mathrm{N})} = \Big((\mathrm{Node_t} \times \mathtt{Integer} \times \mathrm{oid} \times \mathrm{oid}) \\ \times \big((\mathrm{CNode_t} \times \mathtt{Boolean}) \times (\alpha_\mathrm{C})_\perp + \beta_\mathrm{C}\big)_\perp + \beta_\mathrm{N}\Big)\ \mathtt{Object} + \mathit{Values}. \tag{5}$$

We use 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 6 (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 $C_0,\dots,C_{n-1}$ be the supertypes of C_n, i.e, C_i is inherited from C_{i-1}. The class type of C is defined as:

1. $C_i \in \mathfrak{B}, i \in \{0,\dots,n\}$ then
$\mathcal{C}^0_\alpha = \Big(C_0 \times \big(C_1 \times (C_2 \times \cdots \times (C_n \times \alpha_\perp)_\perp)_\perp\big)_\perp\Big)_\perp \in \mathfrak{C}$,
2. $\mathfrak{U}_\mathfrak{C} \supset \mathfrak{C}$, where $\mathfrak{U}_\mathfrak{C}$ is the set of universe types with $\mathcal{C}^0_\alpha := \mathcal{U}^0_\alpha$.

Thus in our example we represent the class type of class `Node` by the HOL type

$$(\alpha_C,\beta_C)\ \mathtt{Node} = \Big((\mathrm{Node_t} \times \mathtt{Integer} \times \mathrm{oid} \times \mathrm{oid}) \\ \times \big((\mathrm{CNode_t} \times \mathtt{Boolean}) \times (\alpha_C)_\perp + \beta_C\big)_\perp\Big)\ \mathtt{Object}. \tag{6}$$

Here, α_C allows for extension with new classes by inheriting from `CNode` while β_C allows for direct inheritance from `Node`.

The outermost $\_{}_\perp$ reflect the fact that class objects may also be undefined, in particular after projecting them from some term in the universe or failing type casts. Thus, also the arguments of constructors may be undefined.

3.2 Elementary Object Construction

For each class, we provide injections and projections into an object universe. In the case of the class `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:

$$\mathrm{mk}^{(0)}_{\mathrm{Object}}\, \mathit{obj} = \mathrm{Inl}\, \mathit{obj} \qquad \text{with type } \alpha\ \texttt{Object} \Rightarrow \mathcal{U}^0_\alpha \tag{7}$$

where *obj* is a variable denoting the current object. Recall that we denote the level on which a constant (e. g., injection, projection, attribute accessor) is defined by a superscript (in parenthesis) and the class for which the constant is defined by a subscript.

The inverse function for projecting an `Object` object out of a universe can be defined as follows:

$$\mathrm{get}^{(0)}_{\mathrm{Object}}\, u = \begin{cases} k & \text{if } u = \mathrm{Inl}\, k \\ \varepsilon k.\, \mathrm{true} & \text{if } u = \mathrm{Inr}\, k \end{cases} \qquad \text{with type } \mathcal{U}^0_\alpha \Rightarrow \alpha\ \texttt{Object}. \tag{8}$$

where $\varepsilon x.\, P x$ is the Hilbert-operator that chooses an arbitrary x satisfying P.

In the general case, the definitions of the injections and projections are a little bit more complex, but follow 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. For example, we define the projector for accessing the `left` attribute of the class `Node`:

$$\mathit{obj}.\mathrm{left}^{(0)} \equiv (\mathrm{fst} \circ \mathrm{snd} \circ \mathrm{snd} \circ \mathrm{fst})\, \ulcorner \mathrm{base}\ \mathit{obj} \urcorner \tag{9}$$

with type $(\alpha_1, \beta)\,\mathrm{Node} \Rightarrow \mathrm{oid}_\bot$ and where base is a variant of snd over lifted tuples:

$$\mathrm{base}\, x \equiv \begin{cases} b & \text{if } x = \llcorner(a,b)\lrcorner, \\ \bot & \text{otherwise.} \end{cases} \tag{10}$$

For attributes with non-value types we return an *oid*. As we return *oid* for all class types, the underlying representation map is not injective and therefore not type-safe. In contrast, in Section 4, we show how these projectors can be used to define a type-safe variant of projectors.

Similarly, we can define injections, or setters, for each attribute. For example, for setting the attribute `left` to a specific value *lft* we define:

$$\begin{aligned} \mathit{obj}.\mathrm{set}^{(0)}_{\mathrm{left}}\, \mathit{lft} \equiv\ & \mathrm{let} \\ & \quad \mathit{oid} = \mathrm{OidOf}\, \mathit{obj} \\ & \quad i = \mathit{obj}.\mathrm{i}^{(0)} \\ & \quad \mathit{right} = \mathit{obj}.\mathrm{right}^{(0)} \\ & \quad \mathit{ext} = \mathit{obj}.\mathrm{ext}^{(0)} \\ & \mathrm{in} \\ & \quad \llcorner((\mathrm{Object}_\mathrm{t}, \mathit{oid}), \llcorner((\mathrm{Node}_\mathrm{t}, i, \mathit{lft}, \mathit{right}), \mathit{ext})\lrcorner)\lrcorner \end{aligned} \tag{11}$$

where OidOf _ accesses the object identifier, $\_.\mathrm{i}^{(0)}$ and $\_.\mathrm{right}^{(0)}$ are attribute accessors, and $\_.\mathrm{ext}^{(0)}$ is an accessor for possible extension of the object, i. e., parts defined in subclasses, e. g., `CNode`.

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:

$$\mathrm{isUniv}^{(0)}_{\mathrm{Object}}\ u = \begin{cases} \mathrm{true} & \text{if } u = \mathrm{Inl}\,k \\ \mathrm{false} & \text{if } u = \mathrm{Inr}\,k \end{cases} \qquad \text{with type } \mathcal{U}^0_\alpha \Rightarrow \mathrm{bool}. \tag{12}$$

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, 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 (using def_), e. g., not equal to $\bot$:

$$\mathrm{isKind}^{(0)}_{\mathrm{Object}}\ obj = \mathrm{def}\ obj \qquad \text{with type } \alpha\ \texttt{Object} \Rightarrow \mathrm{bool}. \tag{13}$$

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

$$\mathrm{isType}^{(0)}_{\mathrm{Object}}\ obj = \mathrm{isKind}^{(0)}_{\mathrm{Object}}\ obj \wedge \neg\big((\mathrm{def} \circ \mathrm{base})\ obj\big) \ \text{with type } \alpha\ \texttt{Object} \Rightarrow \mathrm{bool}. \tag{14}$$

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

Finally, we define casts, i. e., ways to convert 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 handled by polymorphism. Therefore we can define the type casts as simple compositions of projections and injections, e. g.:

$$\mathrm{Node}^{(0)}_{[\mathrm{Object}]} = \mathrm{get}^{(0)}_{\mathrm{Object}} \circ \mathrm{mk}^{(0)}_{\mathrm{Node}} \quad \text{with type } (\alpha_1, \beta_1)\,\mathrm{Node} \Rightarrow (\alpha_1, \beta_1)\ \texttt{Object}, \tag{15}$$

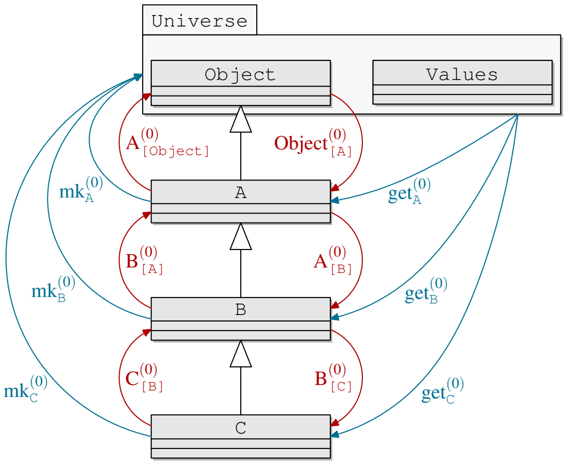
$$\mathrm{Object}^{(0)}_{[\mathrm{Node}]} = \mathrm{get}^{(0)}_{\mathrm{Node}} \circ \mathrm{mk}^{(0)}_{\mathrm{Object}} \quad \text{with type } (\alpha_1, \beta_1)\ \texttt{Object} \Rightarrow (\alpha_1, \beta_1)\,\mathrm{Node}. \tag{16}$$

These type-casts are changing the *static type* of an object, while the *dynamic type* remains unchanged. Figure 3 summarizes this construction for the three classes `A`, `B`, and `C`.

Note, for a universe construction without values, e. g., $\mathcal{U}^0_\alpha = \alpha\ \texttt{Object}$, 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, one can understand the projections as a cast from the universe type to the given class type and the injections as the inverse cast operation.

Now, if we build a theorem over class invariants (based finally on these projections, injections, casts, characteristic sets, etc.), it will remain valid even if we extend the universe via α and β instantiations. Therefore, we have solved the problem of structured extensibility for object-oriented languages.

This construction establishes a subtype relation via inheritance. Therefore, a set of Nodes (with type $\big((\alpha_1, \beta)\,\mathrm{Node}\big)\ \texttt{Set}$) can also contain objects of type `CNode`. For resolving operation overriding, i. e., late-binding, the packages generates operation tables for user-defined operations; see [12,10] for details.



Universe
Object
Values
A
B
C
$A^{(0)}_{[\texttt{Object}]}$
$\text{Object}^{(0)}_{[\texttt{A}]}$
$B^{(0)}_{[\texttt{A}]}$
$A^{(0)}_{[\texttt{B}]}$
$C^{(0)}_{[\texttt{B}]}$
$B^{(0)}_{[\texttt{C}]}$
$\text{mk}^{(0)}_{\texttt{A}}$
$\text{mk}^{(0)}_{\texttt{B}}$
$\text{mk}^{(0)}_{\texttt{C}}$
$\text{get}^{(0)}_{\texttt{A}}$
$\text{get}^{(0)}_{\texttt{B}}$
$\text{get}^{(0)}_{\texttt{C}}$

Figure 3 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`).

3.3 Properties of Elementary Objects

Based on the presented definitions, our object-oriented datatype package proves that our encoding of object structures is a faithful representation of object-orientation (e. g., in the sense of languages like Java or Smalltalk or the UML standard [2]). These theorems are proven for each model, e. g., during loading of a specific class model. This is similar to other datatype packages in interactive theorem provers. Further, these theorems are also a prerequisite for successful reasoning over object structures.

In the following, we assume an arbitrary model with the classes `A`, `B` and `C` where `B` is a subclass of `A` and `C` is a subclass of `B` (recall Figure 3). We start by proving this subtype relation for both our class type and universe type representation:

$$\frac{\text{isUniv}^{(0)}_{\text{A}}\ \mathit{univ}}{\text{isUniv}^{(0)}_{\text{B}}\ \mathit{univ}} \tag{17a}$$

$$\frac{\text{isType}^{(0)}_{\text{B}}\ \mathit{obj}}{\text{isKind}^{(0)}_{\text{A}}\ \mathit{obj}} \tag{17b}$$

We also show that we can switch between the universe representations and object representation without losing information, in fact, both type systems are isomorphic:

$$\frac{\text{isUniv}^{(0)}_{\text{A}}\ \mathit{univ}}{\text{mk}^{(0)}_{\text{A}}\ (\text{get}^{(0)}_{\text{A}}\ \mathit{univ}) = \mathit{univ}} \tag{18a}$$

$$\frac{\text{isType}^{(0)}_{\text{A}}\ \mathit{obj}}{\text{get}^{(0)}_{\text{A}}\ (\text{mk}^{(0)}_{\text{A}}\ \mathit{obj}) = \mathit{obj}} \tag{18b}$$

$$\frac{\text{isType}^{(0)}_{\text{B}}\ obj}{\text{isUniv}^{(0)}_{\text{A}}(\text{mk}^{(0)}_{\text{A}}\ obj)} \tag{19a}$$

$$\frac{\text{isUniv}^{(0)}_{\text{A}}\ univ}{\text{isType}^{(0)}_{\text{A}}(\text{get}^{(0)}_{\text{A}}\ univ)} \tag{19b}$$

Moreover, we can "re-cast" an object safely, i. e., up and down casts are idempotent. However, casting an object deeper in the subclass hierarchy than its dynamic type results in undefinedness:

$$\frac{\text{isType}^{(0)}_{\text{A}}\ obj}{obj^{(0)}_{[\text{B}]} = \bot} \tag{20a}$$

$$\frac{\text{isType}^{(0)}_{\text{B}}\ obj}{\big((obj^{(0)}_{[\text{A}]})^{(0)}_{[\text{B}]}\big) = obj} \tag{20b}$$

and also, the cast operations are strict and transitive, e. g.:

$$\frac{}{\bot^{(0)}_{[\text{A}]} = \bot} \tag{21a}$$

$$\frac{\text{isType}^{(0)}_{\text{C}}\ obj}{(obj^{(0)}_{[\text{B}]})^{(0)}_{[\text{A}]} = obj^{(0)}_{[\text{A}]}} \tag{21b}$$

Further, for all class types c, both $\text{isType}^{(0)}_{\text{c}}\ \bot = \texttt{false}$ and $\text{isKind}^{(0)}_{\text{c}}\ \bot = \texttt{false}$ are valid.

Summarizing, these derived rules show that our encoding of inheritance establishes a subtype relation. Moreover, the (informal) relations between classes one would expect from languages like UML, Java, or Spec#, also hold in our encoding.

Our datatype package also derives similar properties for the injections and projections into attributes automatically. For example, assume the class `A` has two attributes `a` and `b` then we derive among others:

$$\frac{obj \neq \bot}{(obj.\,\text{set}^{(0)}_{\text{a}}\ x).\,\text{a}^{(0)} = x} \tag{22a}$$

$$\frac{}{(obj.\,\text{set}^{(0)}_{\text{a}}\ x).\,\text{b}^{(0)} = obj.\,\text{b}^{(0)}} \tag{22b}$$

$$\frac{}{(obj.\,\text{set}^{(0)}_{\text{a}}\ x).\,\text{set}^{(0)}_{\text{a}}\ y = obj.\,\text{set}^{(0)}_{\text{a}}\ y} \tag{23}$$

$$\frac{}{(obj.\,\text{set}^{(0)}_{\text{a}}\ x).\,\text{set}^{(0)}_{\text{b}}\ y = (obj.\,\text{set}^{(0)}_{\text{b}}\ y).\,\text{set}^{(0)}_{\text{a}}\ x} \tag{24}$$

As we use a shallow embedding of object-oriented data structures into HOL, these properties cannot be proven as meta-theoretic properties of our encoding. Instead, our datatype package proves these properties, fully automatically, during the import of an object-oriented data model.

4 Level 1: A First Type-safe Embedding of COOL

In this section, we present a type-safe embedding of the accessors of COOL. As a prerequisite, we define the *store* as a partial map based on the concept of object universes:

$$\alpha\,\mathrm{St} := \mathrm{oid} \rightharpoonup \mathscr{U}_\alpha\,. \tag{25}$$

Since all operations over our object store will be parametrized by $\alpha\,\mathrm{St}$, we introduced the following type synonym:

$$V_\alpha(\tau) := \alpha\,\mathrm{St} \Rightarrow \tau\,. \tag{26}$$

Thus we can define type-safe accessor functions: object identifiers (references) are completely encapsulated, i. e., on level 1 no object identifiers are visible. For example, the function for accessing the left attribute of an object of class Node in a system state σ works by taking the object, projecting the oid for the `left` attribute, de-reference it in the state σ (which gives a value in the current universe), and project from this the class object of type `Node`. Formally, this is expressed as follows:

$$obj.\mathrm{left}^{(1)} \equiv \lambda\,\sigma.\begin{cases} \mathrm{get}^{(0)}_{\mathrm{Node}}\,u & \text{if } \sigma(obj.\mathrm{left}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.}\end{cases} \tag{27}$$

For set- or sequence valued accessors, we have to provide definitions that de-reference each element of, e. g., a set of object identifiers and build a set of typed objects.

The type of the object-language accessor `.left` returning an object of type `Node`, which is in fact a function of type `Node` $\to$ `Node`, is now represented by our construction as follows:

$$\_.\,\mathrm{left}^{(1)} :: V_{(\alpha_C,\beta_C)}((\alpha_C,\beta_C)\ \texttt{Node}) \Rightarrow V_{(\alpha_C,\beta_C)}((\alpha_C,\beta_C)\ \texttt{Node})\,. \tag{28}$$

Thus, the representation map is injective on types; subtyping is represented by type instantiation on the HOL-level. However, due to our universe construction, the theory on accessors, casts, etc. is also extensible.

All other operations like casting, type- or kind-check are lifted as follows:

$$obj^{(1)}_{[\mathrm{A}]} \equiv \lambda\,\sigma.\ (obj\,\sigma)^{(0)}_{[\mathrm{A}]} \tag{29}$$

$$\mathrm{isType}^{(1)}_{\mathrm{A}}\,obj \equiv \lambda\,\sigma.\ \mathrm{isType}^{(0)}_{\mathrm{A}}\,(obj\,\sigma) \tag{30}$$

$$\mathrm{isKind}^{(1)}_{\mathrm{A}}\,obj \equiv \lambda\,\sigma.\ \mathrm{isKind}^{(0)}_{\mathrm{A}}\,(obj\,\sigma) \tag{31}$$

$$\partial\,obj \equiv \lambda\,\sigma.\ \mathrm{def}(obj\,\sigma) \tag{32}$$

Their types are analogously lifted as in the accessor as discussed above.

In the following, we show how the other operations of COOL can be represented by HOL operators (for simplicity, we will use overloading of operator symbols). Constant symbols in COOL like 0,1,2,…, true,false, $\{\}$ will be represented by constant functions that just drop the state: $\lambda\,\sigma.0$,$\lambda\,\sigma.1$,$\lambda\,\sigma.2$,…, $\lambda\,\sigma.\mathrm{true}$,$\lambda\,\sigma.\mathrm{false}$, $\lambda\,\sigma.\{\}$.

All operators of the COOL language are just "lifted" from their HOL counterparts by passing the implicit state argument, for example, the case of the binary operators is covered by:

$$X\,op\,Y = \lambda\,\sigma.\,(X\,\sigma)\,op\,(Y\,\sigma) \tag{33}$$

where op stands for $\_ = \_, \_\wedge\_, \_\rightarrow\_, \_+\_, \_*\_, \_\in\_, \_\cup\_, \ldots$.

If a path expression is used as an expression, i. e., as argument of a COOL operator as in $\mathit{self}.\mathrm{i}^{(1)} > 5$, for example, it is implicitly dropped: $(\lambda\,\sigma.\ulcorner(\mathit{self}.\mathrm{i}^{(1)})\sigma\urcorner) > 5$. This is motivated by turning COOL in a conventional HOL-language without exceptional elements (similar to Spec# or JML); these elements were strictly propagated within path expressions, can be tested via $\partial\,obj$, and will be interpreted by arbitrary, underspecified values when passed as arguments to operators.

The lifted operations, for example $\_\wedge\_$, have the type $V_\alpha(\mathrm{bool}) \Rightarrow V_\alpha(\mathrm{bool}) \Rightarrow V_\alpha(\mathrm{bool})$ if the corresponding operator $\_\wedge\_$ of the meta-language HOL has the type $\mathrm{bool} \Rightarrow \mathrm{bool} \Rightarrow \mathrm{bool}$. This exemplifies again the injectivity of the representation map on associated types.

The judgment $\sigma \models E$ is simply a shortcut for $E\,\sigma$. As a consequence of these definitions, $\sigma \models E \wedge E'$ is just equivalent to $\sigma \models E \wedge \sigma \models E'$ (recall that we use overloading).

An alternative semantic choice for the semantics of COOL consists in strict extensions of the operators and a three-valued logic; a further subdivision here is a strict logic (cf. [14]) or a Strong Kleene Logic (cf. [12]). Such an alternative allows for a uniform handling of exceptions (like illegal memory access or "1 div 0") as occurs in programming languages like Java. The key difference to the semantics of COOL as presented above is the definition for the operator family:

$$X\,op\,Y = \lambda\,\sigma.\begin{cases} \llcorner\ulcorner X\,\sigma\urcorner\,op\,\ulcorner Y\,\sigma\urcorner\lrcorner & \text{if } \mathrm{def}(X)\wedge\mathrm{def}(Y),\\ \bot & \text{otherwise}\end{cases} \tag{34}$$

where the case of monadic operators and constants is handled analogously. Another difference is the injection of paths into expressions, which is just the identity in this variant. However we will not further discuss this alternative here since it is clearly too far away from the mainstream in object-oriented program verification.

Finally, the properties of the previous section can be rephrased for level 1. The "lifted versions" of these rules will have to take the store σ into account. This results in two different patterns shown as follows:

$$\frac{\sigma \models \mathrm{isType}^{(1)}_{\mathrm{A}}\,obj}{\sigma \models \mathrm{isKind}^{(1)}_{\mathrm{A}}\,obj} \tag{35a}$$

$$\frac{\sigma \models \mathrm{isType}^{(1)}_{\mathrm{C}}\,obj}{(obj^{(0)}_{[\mathrm{B}]})^{(1)}_{[\mathrm{A}]}\ \sigma = obj^{(1)}_{[\mathrm{A}]}\ \sigma} \tag{35b}$$

5 Level 2: Co-inductive Properties in Object Structures

A main contribution of our work is the encoding of co-inductive properties over object structures, including the support for class invariants.

Recall our running example, i. e., Figure 1, where the class `Node` describes a potentially infinite cyclic object structure. Later, we will give semantics to the two invariants `positive` and `flip`. Since they represent recursive predicates, they must be defined conservatively via greatest fixed-points. The Isabelle library also offers a theory for the greatest fixed-point operator $\mathrm{gfp} :: (\alpha\,\mathrm{set} \Rightarrow \alpha\,\mathrm{set}) \Rightarrow \alpha\,\mathrm{set}$. For technical reasons, we must talk over *characteristic sets*, e. g., Node_positive_Set, instead of invariants, e. g., Node_positive, although both are closely connected:

$$\begin{aligned}\mathrm{Node\_positive}(\mathit{self}) &\equiv \mathit{self} \in \mathrm{Node\_positive\_Set}\\ &\equiv \mathit{self} \in (\lambda\,\sigma.\ \mathrm{gfp}(\mathrm{Node\_positive\_F}\ \sigma))\end{aligned}$$

We pick the example definition Node_positive_F as core-prerequisite for the invariant; recall that this invariant requires from the attribute `i` of class `Node` to have values greater than `5`:

$$\begin{aligned}\mathrm{Node\_positive\_F} :: {}& \mathscr{U}^1_{(\alpha_\mathrm{C},\beta_\mathrm{C},\beta_\mathrm{N})}\,\mathrm{St} \Rightarrow (\mathscr{U}^1_{(\alpha_\mathrm{C},\beta_\mathrm{C},\beta_\mathrm{N})}\,\mathrm{St} \Rightarrow (\alpha_C,\beta_C)\ \texttt{Node}\,\mathrm{set})\\ &\Rightarrow (\mathscr{U}^1_{(\alpha_\mathrm{C},\beta_\mathrm{C},\beta_\mathrm{N})}\,\mathrm{St} \Rightarrow (\alpha_C,\beta_C)\ \texttt{Node}\,\mathrm{set})\\ \mathrm{Node\_positive\_F} \equiv {}& \lambda\,\sigma.\ \lambda\,X.\ \Big\{\mathit{self}\ \Big|\ \sigma \vDash \partial\,\mathit{self}.\,\mathrm{i}^{(1)} \wedge \sigma \vDash \mathit{self}.\,\mathrm{i}^{(1)} > 5\\ &\wedge\ \sigma \vDash \partial\,\mathit{self}.\,\mathrm{left}^{(1)} \longrightarrow\ \sigma \vDash \mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\mathit{self}.\,\mathrm{left}^{(1)}\\ &\qquad \wedge \sigma \vDash (\mathit{self}.\,\mathrm{left}^{(1)}) \in X\\ &\wedge\ \sigma \vDash \partial\,\mathit{self}.\,\mathrm{right}^{(1)} \longrightarrow\ \sigma \vDash \mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\mathit{self}.\,\mathrm{right}^{(1)}\\ &\qquad \wedge \sigma \vDash (\mathit{self}.\,\mathrm{right}^{(1)}) \in X\Big\}\end{aligned} \tag{36}$$

The first part of this (generated) definition is a straightforward translation of the formula in the **`inv`** part of the class declaration in Figure 1; the clauses $\sigma \vDash \partial\,\mathit{self}.\,\mathrm{left}^{(1)} \longrightarrow \sigma \vDash \mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\mathit{self}.\,\mathrm{left}^{(1)} \wedge \sigma \vDash (\mathit{self}.\,\mathrm{left}^{(1)}) \in X$ (and for `_.right`) are added schematically as a result of the recursive type definitions.

For a given invariant from the input, our package generates definitions like the one shown above and derives the following *class invariant theorem* expressing the intuition of our invariant:

$$\begin{aligned}\sigma \vDash \mathrm{Node\_positive}\ \mathit{self} = {}& \sigma \vDash \partial\,\mathit{self}.\,\mathrm{i}^{(1)}\\ &\wedge \sigma \vDash \mathit{self}.\,\mathrm{i}^{(1)} > 5\\ &\wedge\ \sigma \vDash \partial\,\mathit{self}.\,\mathrm{left}^{(1)} \longrightarrow\ \sigma \vDash \mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\mathit{self}.\,\mathrm{left}^{(1)}\\ &\qquad \wedge \sigma \vDash \mathrm{Node\_positive}\ \mathit{self}.\,\mathrm{left}^{(1)}\\ &\wedge\ \sigma \vDash \partial\,\mathit{self}.\,\mathrm{right}^{(1)} \longrightarrow\ \sigma \vDash \mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\mathit{self}.\,\mathrm{right}^{(1)}\\ &\qquad \wedge \sigma \vDash \mathrm{Node\_positive}\ \mathit{self}.\,\mathrm{right}^{(1)}\end{aligned} \tag{37}$$

or equivalently (by extensionality and by COOL notation (see Section 4)):

$$\begin{aligned}\text{Node\_positive}\,\mathit{self} = \partial\,\mathit{self}.\mathrm{i}^{(1)} \wedge \mathit{self}.\mathrm{i}^{(1)} > 5 \\ \wedge\ \partial\,\mathit{self}.\mathrm{left}^{(1)} \to \mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\mathit{self}.\mathrm{left}^{(1)} \\ \wedge\ \text{Node\_positive}\,\mathit{self}.\mathrm{left}^{(1)} \\ \wedge\ \partial\,\mathit{self}.\mathrm{right}^{(1)} \to \mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\mathit{self}.\mathrm{right}^{(1)} \\ \wedge\ \text{Node\_positive}\,\mathit{self}.\mathrm{right}^{(1)}\end{aligned} \tag{38}$$

Since our package supports the use of kind and type tests inside invariants, one can establish useful constraints for verification. For the class Node, we can state the following invariant even on the level of the input:

```
inv positive_type: self.i > 5
                   ∧ ∂ self.left ⟶ isType(1)Node self.left
                   ∧ ∂ self.right ⟶ isType(1)Node self.right
```

This example is in fact a canonical alternative interpretation of the positive invariant: Node_positive leads only to the requirement that sub-graphs have the Node-kind; the above declaration (which leads to structurally the same invariant theorem except that $\mathrm{isKind}^{(1)}_{\mathrm{Node}}\,\_$ predicates were replaced by their $\mathrm{isType}^{(1)}_{\mathrm{Node}}\,\_$ counterparts) strengthens this to the requirement that all reachable nodes have Node-type. We call the characteristic sets of these alternative invariants the *characteristic type set*, while the default is called the *characteristic kind set*.

For kind sets and type sets, the packages proves automatically by monotonicity of the approximation functions and their point-wise inclusion:

$$\sigma \models \text{Node\_positive\_Type\_Set} \subseteq \text{Node\_positive\_Set} \tag{39}$$

This kind of theorem is another example for properties that remain valid if we add further classes in a class model.

Now we relate class invariants of subtypes to class invariants of supertypes. Here, we use cast functions described in the previous section; we write $\mathit{self}^{(1)}_{[\mathrm{Node}]}$ for the object *self* converted to the type Node of its superclass. The trick is done by defining a new approximation for an inherited class CNode on the basis of the approximation function of the superclass:

$$\text{CNode\_flip\_F} \equiv \lambda\,\sigma.\,\lambda\,X. \\ \left\{\mathit{self}\ \middle|\ \left(\mathit{self}^{(1)}_{[\mathrm{Node}]} \in \left(\text{Node\_positive\_F}\,\sigma\,(\lambda\,x.\,x^{(1)}_{[\mathrm{Node}]})\,\grave{}\,X\right)\right) \wedge (\text{CNode\_flip}\,\mathit{self})\right\} \tag{40}$$

where $\_\,\grave{}\,\_$ denotes the point-wise application.

Similar to the work of Berghofer and Wenzel [8] or Paulson [32] we can handle 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.

These type sets have the usual properties that one associates with object-oriented type systems. Let $\mathfrak{C}_N$ ($\mathfrak{K}_N$) be the characteristic type set (characteristic kind set) of a class `N` and let $\mathfrak{C}_C$ and $\mathfrak{K}_C$ be the corresponding type sets of a direct subclass `C` of `N`, then our encoding process proves formally that the characteristic type set is a subset of the kind set, i. e.:

$$\sigma \vDash \mathit{self} \in \mathfrak{C}_N \longrightarrow \sigma \vDash \mathit{self} \in \mathfrak{K}_N\,. \tag{41}$$

Moreover, the kind set of the subclass is (after type cast) a subset of the type set (and thus also of the kind set) of the superclass:

$$\sigma \vDash \mathit{self} \in \mathfrak{K}_C \longrightarrow \sigma \vDash \mathit{self}^{(1)}_{[\mathrm{N}]} \in \mathfrak{C}_N\,. \tag{42}$$

These proofs are co-inductions and involve a kind of bi-simulation of (potentially) infinite object structures. Further, these proofs depend on theorems that are already proven over the pre-defined types, e. g., `Object`. These proofs where done in the context of the initial universe $\mathscr{U}^0$ and can be instantiated directly in the new universe without replaying the proof scripts; this is our main motivation for an *extensible* construction.

On the basis of these definitions, we can now give an alternative, stronger semantic interpretation of COOL. The key issue of this language interpretation is that the definedness of an accessor function implies that the resulting object is indeed *valid*, i. e., satisfies its class invariant. Thus, the static type of an object gets a meaning via the class invariants, not only structurally as in the previous interpretation. The accessors are defined like:

$$\mathit{self}.\mathrm{left}^{(2)} \equiv \lambda\,\sigma.\begin{cases} \mathit{self}.\mathrm{left}^{(1)} & \text{if } \sigma \vDash \mathit{self}.\mathrm{left}^{(1)} \in \mathfrak{K}_{\mathit{Node}} \\ \bot & \text{otherwise.} \end{cases} \tag{43}$$

The key concepts of kind and type are interpreted as follows:

$$\mathrm{isType}^{(2)}_{\mathrm{A}}\,\mathit{self} \equiv \lambda\,\sigma.\,\sigma \vDash \mathit{self} \ \in \mathfrak{C}_A \tag{44}$$

$$\mathrm{isKind}^{(2)}_{\mathrm{A}}\,\mathit{self} \equiv \lambda\,\sigma.\,\sigma \vDash \mathit{self} \ \in \mathfrak{K}_A \tag{45}$$

All other semantic definitions for COOL remain unchanged.

The additional property of this semantic interpretation of COOL just formalizes the intuition already stated above:

$$\frac{\sigma \models \partial(\mathit{self}.\mathrm{left}^{(2)})}{\sigma \models \mathrm{isKind}^{(2)}_{\mathrm{Node}}\,\mathit{self}.\mathrm{left}^{(2)}} \tag{46}$$

With these derived rules, defining *valid states* by requiring that set of objects in the store satisfies the invariant becomes superfluous.

6 A Modular Proof-methodology for Object-oriented Modeling

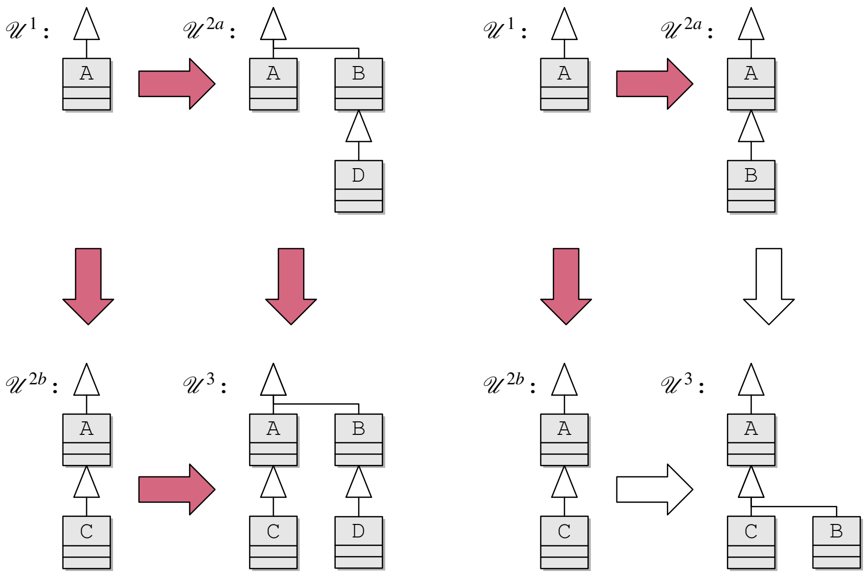
In the previous sections, we discussed a technique to build *extensible* object-oriented data models. Now we turn to the wider goal of a *modular* proof methodology for object-oriented systems and explore achievements and limits of our approach with respect to this goal. Two aspects of modular proofs over object-oriented models have to be distinguished:

1. the modular construction of theories over object-data models, and
2. a modular way to represent dynamic type information or storage assumptions underlying object-oriented programs.

With respect to the former, the question boils down to what degree extensions of class models and theories built over them can be merged. With respect to the latter, we will show how co-inductive properties over the store help to achieve this goal.

6.1 Non-overlapping Merges

The positive answer to the modularity question is that object-oriented data model theories can be merged provided that the extensions to the underlying object-data models are non-overlapping. Two extensions are *non-overlapping*, if their set of classes has no common direct parent class (see Figure 4a). In these cases, there exists a most general type instance of the merged object universe $\mathscr{U}^3$ (the type unifier of both extended universes $\mathscr{U}^{2a}$ and $\mathscr{U}^{2b}$); thus, all theorems built over the extended universes are still valid over the merged universe (see Figure 4a). We claim that the non-overlapping case is the more important one; for example, all libraries of the HOL-OCL system [12] were linked to the examples in its substantial example suite this way. Without extensibility, the datatype package would have to require the recompilation of the libraries,



(a) Non-conflicting Merges (b) Conflicting Merges

Figure 4 Merging Universes: Figure 4a illustrate the non-overlapping extension of a class `A` with one direct subclass `C` and a new hierarchy consisting of the classes `B` and `D` which are rooted on the same level as `A`. In contrast, Figure 4b illustrates a conflicting merge. In this case, the class `A` is extended independently with the direct subclasses `B` and `C` which causes a conflict when merging these two extensions.

which takes in the case of the HOL-OCL system about 20 minutes. In the following, we discuss two approaches for tackling this limitation of our framework.

6.2 Overlapping Merges

Unfortunately, there is an important case in object-oriented modeling that will be considered as an overlap in our package. Consider the case illustrated in Figure 4b. Here, the parent class `A` is in the class set of both extensions (*including* parent classes). The technical reason for the conflict is that the order of insertions of subclasses into a parent class is relevant since the type sum $\alpha + \beta$ is not a commutative and associative type constructor.

In our encoding scheme of object-oriented data models, this scenario of extensions represents an overlap that the user is forced to resolve. One possibility is to arrange an order of the extensions by changing the import hierarchy of theories producing overlapping extensions. This worst case results in re-running the unmodified proof scripts of either `B` or `C`. Another option is to resolve the (potential) conflict in advance by introducing an empty class `B'` and let inherit `B` from there. A further option consists in adding a mechanism into our package allowing to specify for a child-class the position in the insertion-order.

6.3 Modularity in an Open World: Dynamic Typing

Our notion of extensible class models generalizes the distinction "open world assumption" vs. "closed world assumption" widely used in object-oriented modeling. Our universe construction is strictly "open world" by default; the case of a "closed world" results from instantiating all α- and β-variables ("holes") in the universe by the unit type. Since such an instantiation can also be partial, there is a spectrum between both extremes. Furthermore, one can distinguish *α-finalizations*, i. e., instantiation of a α-variable in the universe by the unit type, and *β-finalizations*. The former closes a class hierarchy with respect to subtyping, the latter prevent that a parent class may have further direct children (which makes the automatic derivation of an exhaustion lemma for this parent class possible).

In this and the subsequent section, we consider an extension of path expressions in the COOL language by (side-effect free) methods. In usual object-oriented languages, methods can be overridden, method invocations like in object-oriented languages require a mechanism for the resolution of overridden methods such as *late binding* as used in Java. Late binding uses the dynamic type $\mathrm{isType}_{\mathrm{X}}^{(1)}$ *self* of *self*. The late-binding method invocation is notorious for its difficulties for modular proof. Consider the case of an operation:

```
method Node::m():Boolean
  pre:  P
  post: Q
```

Assume that the implementation of `m` invokes itself recursively, e. g., by `self.left.m()`. Based on an open world assumption, the postcondition `Q` cannot be established in general since it is unknown which concrete implementation is called at the invocation.

Based on our universe construction, there are two ways to cope with this underspecification. First, finalizations of all child classes of `Node` results in a *partial* closed world assumption allowing to treat the method invocation as case switch over dynamic types and direct calls of method implementations. Second, similarly to the co-inductive invariant example in Section 5 which ensures that a specific dereferentiation is in fact defined, we can specify that a specific de-referentiation $\mathit{self}.\,\mathrm{left}^{(1)}$ has a given dynamic type. An analogous invariant $\mathrm{Inv}_{\mathrm{left}}(\mathit{self})$ can be defined coinductively. From this invariant, we can directly infer facts like $\mathrm{isType}^{(1)}_{\mathrm{Node}}\,(\mathit{self}.\,\mathrm{left}^{(1)})$, and $\mathrm{isType}^{(1)}_{\mathrm{Node}}\,(\mathit{self}.\,\mathrm{left}^{(1)}.\,\mathrm{left}^{(1)})$, i. e., in an object graph satisfying this invariant, the left “spine” must consist of nodes of dynamic type `Node`. Strengthening the precondition `P` by $\mathrm{Inv}_{\mathrm{left}}(\mathit{self})$ consequently allows to establish postcondition `Q`—in a modular manner, since only the method implementation above has to be considered in the proof. Invoking the method on an object graph that does not satisfy this invariant can therefore be considered as a breach of the contract.

6.4 Modularity in an Open World: Storage Assumptions

Similarly to co-inductive invariants, it is possible via co-recursive functions to map an object to the set of objects that can be reached along a particular path set. The definition must be co-recursive, since object structures may represent a graph. However, the presentation of this function may be based on a primitive-recursive approximation function depending on a factor $k :: \mathrm{nat}$ that computes this object set only up to the length k of the paths in the path set:

$$\begin{aligned}
&\mathrm{ObjSetA}_{\mathrm{left}}\ 0\,\mathit{self}\,\sigma = \{\}\\
&\mathrm{ObjSetA}_{\mathrm{left}}\ k\,\mathit{self}\,\sigma = \text{if } \sigma \not\models \partial\,\mathit{self} \text{ then } \{\}\\
&\qquad\qquad\qquad\qquad\text{else } \{\mathit{self}\} \cup \mathrm{ObjSetA}_{\mathrm{left}}\,(k-1)\,(\mathit{self}.\,\mathrm{left}^{(1)}\ \sigma)\,\sigma
\end{aligned} \tag{47}$$

The function $\mathrm{ObjSet}_{\mathrm{left}}\ \mathit{self}\,\sigma$ can then be defined as limit

$$\bigcup\nolimits_{n\in\mathrm{Nat}} \mathrm{ObjSetA}_{\mathrm{left}}\ n\,\mathit{self}\,\sigma\,. \tag{48}$$

Moreover, we can add an *internal state invariant*, constraining our concept of state, by using a type definition $\alpha\,\mathrm{St} = \{\sigma :: \mathrm{oid} \rightharpoonup \mathscr{U}^{\alpha} \mid \mathrm{Inv}\ \sigma\}$. Here, we require for Inv that each oid points to an object that contains itself:

$$\forall \mathrm{oid} \in \mathrm{dom}\ \sigma.\ \mathrm{OidOf}(\ulcorner \sigma\,\mathrm{oid} \urcorner) = \mathrm{oid} \tag{49}$$

As a consequence, there exists a “one-to-one”-correspondence between objects and their oid in a state. Thus, sets of objects can be viewed as sets of references, too, which paves the way to interpret these reference sets in different states and specify that an object did not change during a system transition or that there are no references from one object structure into some critical part of another object structure.

7 Application: A Shallow Embedding of IMP++

In the following, we integrate the assertion language COOL for object-oriented data models into a derived Hoare calculus for a small, non-trivial object-oriented language. This language is pretty much in the spirit of Featherweight Java [18], in the sense that it is reduced to the absolute minimum. IMP++ does not even comprise the concept of a method invocation or a procedure call; on the other hand, it provides a "generic slot" for these concepts via the `CMD`-construct, that allows for an arbitrary transition over the entire program state. Given the dynamic type tests of the data model, it is straightforward to *define* an arbitrary overload resolution within this slot. However, demonstrating how this definition scales up with the presented machinery to a modular proof system for methods and their invocation, is a far more evolved subject that we consider beyond the scope of this paper.

Instead, we focus on how our type-safe framework pays off by simplifying the proof rules and consequently the proofs. For example, no side-conditions are necessary related to well-formedness of objects, the syntactic admissibility of attribute accesses of an object or to reasoning along the class hierarchy as in the deep embedding of, e. g., NanoJava [31].

We will show that a compact Hoare logic can be derived from a denotational semantics for IMP++. As the basis, we use IMP [27], a canonical imperative core language available in the Isabelle/HOL library; this language has been inspired by a standard textbook on program semantics [34]. We will extend IMP with object-oriented types, object creation and object update, and a simple form of exceptional computation motivated by illegal memory accesses. Finally, we present a small program that establishes the class invariant of our `CNode` example—although no single transition (i. e., single command) of IMP++ can establish it.

In contrast to the previous sections where definitions and proofs were done automatically for all classes and attributes, the proofs presented in this section are created interactively. However, the rules for the Hoare logic are proven once and for all, and the derivation of the rules for update and create could be automated.

7.1 Syntax

The syntax of IMP++ is introduced via a datatype definition:

$$\begin{aligned} \alpha\,\mathrm{com} = {} & \mathtt{SKIP} && \mid \mathtt{EXN} \\ & \mid \mathtt{CMD}\,\alpha\,\mathrm{cmd} && \mid \mathtt{IF}\,\alpha\,\mathrm{bexp}\,\mathtt{THEN}\,\alpha\,\mathrm{com}\,\mathtt{ELSE}\,\alpha\,\mathrm{com} \\ & \mid \alpha\,\mathrm{com}\,;\alpha\,\mathrm{com} && \mid \mathtt{WHILE}\,\alpha\,\mathrm{bexp}\,\mathtt{DO}\,\alpha\,\mathrm{com} \end{aligned} \tag{50}$$

`SKIP` denotes the empty, successfully terminating command, `EXN` the program that raises an exception (IMP++ possesses only one). The generic command `CMD` takes as argument a function $\alpha\,\mathrm{cmd}$ which is a synonym for a function α state $\Rightarrow$ α $\mathrm{state}_\perp$. Thus, a $\alpha\,\mathrm{cmd}$ is allowed to raise an exception; in our context, this will be used to react operationally on undefined argument-oid's of creation and update operations. The sequential composition, the conditional and the while loop are the conventional constructs of the language. The latter two are controlled by a Boolean expression $\alpha\,\mathrm{bexp}$ which is a synonym for α state $\Rightarrow$ bool (resp. α state $\Rightarrow$ $\mathrm{bool}_\perp$ in the case of a strict version of COOL, as used in [14]). Any COOL expression has a type which is an instance of α bexp, thus, it can be also used as control expression in IMP++.

7.2 Denotational Semantics

The denotational semantics of an imperative language is a relation on states; since uncaught exceptions may occur on the command level, we have also *error states* denoted by $\perp$. Thus, the type of the relation is $(\alpha\ \text{state}_\perp \times \alpha\ \text{state}_\perp)set$. As a consequence, we need as a prerequisite the "strict composition" $\_ \circ_\perp \_$ of relations, of type $(\beta_\perp \times \gamma_\perp)\,\text{set} \Rightarrow (\alpha_\perp \times \beta_\perp)\,\text{set} \Rightarrow (\alpha_\perp \times \gamma_\perp)\,\text{set}$ on relations:

$$\begin{aligned} r \circ_\perp s \equiv \{(\perp,\perp)\} \cup \{(x,z) \mid \operatorname{def} x \wedge (\exists y.\ \operatorname{def} y \wedge (x,y) \in s \wedge (y,z) \in r)\} \\ \cup \{(x,z) \mid \operatorname{def} x \wedge (\exists y.\ \neg \operatorname{def} y \wedge (x,y) \in s \wedge z = \perp)\} \end{aligned} \quad (51)$$

The definition of the semantic function C is a primitive recursion over the syntax:

$$C(\texttt{SKIP}) = \text{Id} \quad (52a)$$

$$C(\texttt{EXN}) = \{(s,t) \mid t = \perp\} \quad (52b)$$

$$C(\texttt{CMD}\ f) = \{(s,t) \mid s = \perp \wedge t = \perp\} \cup \{(s,t) \mid \operatorname{def} s \wedge t = f \ulcorner s \urcorner\} \quad (52c)$$

$$C(c_0;c_1) = C(c_1) \circ_\perp C(c_0) \quad (52d)$$

$$\begin{aligned} C(\texttt{IF}\, b\, \texttt{THEN}\, c_1\, \texttt{ELSE}\, c_2) = {} & \{(s,t) \mid s = \perp \wedge t = \perp\} \\ & \cup \{(s,t) \mid \operatorname{def} s \wedge b \ulcorner s \urcorner = \text{true} \wedge (s,t) \in C c_1\} \\ & \cup \{(s,t) \mid \operatorname{def} s \wedge b \ulcorner s \urcorner = \text{false} \wedge (s,t) \in C c_2\} \end{aligned} \quad (52e)$$

$$C(\texttt{WHILE}\, b\, \texttt{DO}\, c) = \operatorname{lfp}(\Gamma\, b\, (C c)) \quad (52f)$$

where Γ is the usual approximation functional for the least fixed-point operator lfp, enriched by the cases for undefined states:

$$\begin{aligned} \Gamma\, b c d \equiv (\lambda\, \phi.\, \{(s,t) \mid s = \perp \wedge t = \perp\} \cup \{(s,t) \mid \operatorname{def} s \wedge b \ulcorner s \urcorner = \text{true} \wedge (s,t) \in (\phi \circ_\perp c d)\} \\ \cup \{(s,t) \mid \operatorname{def} s \wedge b \ulcorner s \urcorner = \text{false} \wedge s = t\}) \end{aligned} \quad (53)$$

7.3 A Derived Hoare Logic

In our setting, assertions are functions $\alpha :: \text{bot state}_\perp \Rightarrow \text{bool}$. The validity of a Hoare triple is stated as traditional:

$$\models \{P\}c\{Q\} \equiv \forall s\, t.\, (s,t) \in C(c) \longrightarrow P s \longrightarrow Q t \quad (54)$$

Based on the definition for C, we can derive a Hoare calculus for IMP++. Since we focus on correctness proof and not completeness, we present the rules for validity of $\models \_$ directly, avoiding a detour via a derivability notion $\vdash$. Moreover, we use the abbreviation $\odot P$ for $\lambda\, \sigma.\, \operatorname{def} \sigma \wedge P \sigma$. Thus, assertions like $\models \{\odot P'\}c\{\odot Q'\}$ relate "non-exception" states allowing inference of normal behavior. The derived calculus is now surprisingly standard (see Table 1).

$$\frac{}{\vDash \{\odot P\}\,\texttt{SKIP}\{\odot P\}} \tag{55a}$$

$$\frac{\forall s.\, P'\, s \longrightarrow P\, s \quad \vDash \{P\}c\{Q\} \quad \forall s.\, Q\, s \longrightarrow Q'\, s}{\vDash \{P'\}c\{Q'\}} \tag{55b}$$

$$\frac{\vDash \{\odot P\}c\{\odot Q\} \quad \vDash \{\odot Q\}d\{\odot R\}}{\vDash \{\odot P\}c;d \quad \{\odot R\}} \tag{56a}$$

$$\frac{\vDash \{\odot \lambda\, \sigma. P\sigma \wedge (\ulcorner\sigma\urcorner \vDash b)\}c\{\odot P\}}{\vDash \{\odot P\}\{\texttt{WHILE}\}b\{\texttt{DO}\}c\{\odot \lambda\, \sigma. P\sigma \wedge (\ulcorner\sigma\urcorner \vDash \neg b)\}} \tag{56b}$$

$$\frac{}{\vDash \{\lambda\, \sigma.\sigma = \bot\}\, c\, \{\lambda\, \sigma.\sigma = \bot\}} \tag{57a}$$

$$\frac{}{\vDash \{\odot \lambda\, \sigma. \ulcorner\sigma\urcorner \vDash \partial f \wedge Q(f\ulcorner\sigma\urcorner)\}\,\texttt{CMD}\, f\{\odot Q\}} \tag{57b}$$

$$\frac{\vDash \{\odot \lambda\, \sigma. (P\sigma) \wedge (\ulcorner\sigma\urcorner \vDash b) \wedge (\ulcorner\sigma\urcorner \vDash \partial b)\}c\{\odot Q\} \quad \vDash \{\odot \lambda\, \sigma. (P\sigma) \wedge (\ulcorner\sigma\urcorner \vDash \neg b) \wedge (\ulcorner\sigma\urcorner \vDash \partial b)\}d\{\odot Q\}}{\vDash \{\odot P\}\,\texttt{IF}\; b\;\texttt{THEN}\; c\;\texttt{ELSE}\; d\{\odot Q\}} \tag{58}$$

Table 1 The Hoare Calculus for IMP++ is derived from the semantic definitions of IMP++.

7.4 Data Model Specific Hoare Rules

Recall our running example depicted in Figure 1. Besides the type-safe accessor functions, we need families of level-1 (store-related) update and creation operations on objects.

The lifting of update operations to level 1 is straightforward:

$$\mathit{self}.\mathrm{set}^{(1)}_{\mathrm{left}}\, E \equiv \lambda\, \sigma.\sigma(\mathrm{OidOf}\mathit{self} := \mathit{self}\, \sigma.\mathrm{set}^{(0)}_{\mathrm{left}}\, (E\, \sigma)) \tag{59}$$

The operation $\_\,(\_ := \_)$ denotes the usual update on functions. Instead of $\texttt{CMD}(\mathit{self}.\mathrm{set}^{(1)}_{\mathrm{left}}\, E)$ we write $\mathit{self}.\mathrm{left} := E$.

With respect to the creation operations, we define later:

$$\mathrm{newOid}\, \sigma \equiv \varepsilon x. x \notin \mathrm{dom}\, \sigma \tag{60}$$

where $\varepsilon x. P\, x$ is the Hilbert-operator that chooses an arbitrary x satisfying P.

$$\mathrm{new}_{\mathrm{Node}}\, \mathit{oid} \equiv {}_{\llcorner}((\mathrm{Object}_{\mathrm{t}}, \mathit{oid}), {}_{\llcorner}((\mathrm{Node}_{\mathrm{t}}, \bot, \bot, \bot, \bot)_{\lrcorner})_{\lrcorner} \tag{61}$$

The creation operation generates a new object of some type and stores the reference to it in a given attribute of *self*:

$$\begin{aligned} \mathit{self}.\mathrm{new}^{(1)}_{\mathrm{Node[left]}} \; &\equiv \lambda\, \sigma.\ \mathrm{let}\, \sigma' = \sigma(\mathrm{newOid}\, \sigma := \mathrm{new}_{\mathrm{Node}}\, (\mathrm{newOid}\, \sigma)) \\ &\quad \mathrm{in}\, \mathit{self}.\mathrm{set}^{(1)}_{\mathrm{left}}\, (\mathrm{new}_{\mathrm{Node}}\, (\mathrm{newOid}\, \sigma))\sigma' \end{aligned} \tag{62}$$

Instead of $\texttt{CMD}(\mathit{self}.\mathrm{new}^{(1)}_{\mathrm{Node[left]}}\,)$ we write $\mathit{self}.\mathrm{left} := \mathrm{new(Node)}$.

From these definitions, the following family of class model-specific Hoare rules is derived (as usual, we pick the case for attribute left):

$$\frac{}{\models \{\odot \lambda\, \sigma.(\ulcorner\sigma\urcorner \vDash (\partial\, \mathit{self})) \wedge Q(\mathit{self}.\mathrm{set}^{(1)}_{\mathrm{left}}\, E\ulcorner\sigma\urcorner)\}\mathit{self}.\mathrm{left} := E\{\odot Q\}} \tag{63}$$

The analogous case for the creation is a special case of this rule.

7.5 An Example in IMP++

In a fictive object-oriented language, a program that produces the smallest non-trivial object system satisfying the invariant of class CNode, looks as follows:

```
method CNode m():CNode;
var  H1:CNode;
var  H2:CNode;
begin
   H1:= new(CNode);
   H2:= new(CNode);
   H1.i:= 7;
   H1.color:=true;
   H1.left:=H2;
   H1.right:=H2;
   H2.i:= 9;
   H2.color:=false;
   H2.left:=H1;
   H2.right:=H1;
   return H1
end
```

We cannot represent this method directly in IMP++ due to the lack of syntax. However, we can represent the local variables by extending the underlying class model by a *stack object class for method m* (a terminology also used in the Java language specification), and express pre and post conditions for the body called m_{body} translated one-to-one into IMP++.

The stack-object class class `m_stobj` has the form:

```
class m_stobj
   attributes
      self   : Node
      return : CNode
      H1     : CNode
      H2     : CNode
end
```

i. e., it comprises attributes for the local variables `H1` and `H2` with the previously described types as well as a `return` attribute of type `CNode`. The package will then generate the usual update functions for this class and give semantics to the corresponding assignments in our example program (the return statement is viewed as an update to the `return` attribute). With these preliminaries, the encoding of the body of method `m` is one-to-one.

We want to specify that the program establishes by a sequence of creation and update steps the global invariant. Assuming that the stack object m is defined when the method is called (an assumption that reflects the operational behavior of method invocations), the verification of the body is stated in Isabelle as proof goal follows:

$$\vDash \{\odot\, \lambda\, \sigma.\, \sigma \models \partial\, m\} m_{\text{body}} \{\odot\, \sigma \vDash \text{CNode\_flip\_strong}(m.\,\text{return}^{(1)})\} \tag{64}$$

The interactive proof of this statement proceeds in essentially two phases: First, by several applications of the consequence rule (shown in Equation 56a in Table 1) and the update-rules shown in Equation 59, we accumulate an equation system as assertion:

$$
\begin{aligned}
\sigma \models\ & \partial(m.\mathrm{H1}^{(1)}) \\
& \land\ m.\mathrm{H1.i}^{(1)} = 7 \land\ m.\mathrm{H1.color}^{(1)} = \mathrm{true} \\
& \land\ m.\mathrm{H1.left}^{(1)} = m.\mathrm{H2}^{(1)} \land m.\mathrm{H1.right}^{(1)} = m.\mathrm{H2}^{(1)} \\
& \land\ \partial(m.\mathrm{H2}^{(1)}) \\
& \land\ m.\mathrm{H2.i}^{(1)} = 9 \land\ m.\mathrm{H2.color}^{(1)} = \mathrm{false} \\
& \land\ m.\mathrm{H2.left}^{(1)} = m.\mathrm{H1}^{(1)} \land m.\mathrm{H2.right}^{(1)} = m.\mathrm{H1}^{(1)} \\
& \land\ m.\mathrm{return}^{(1)} = m.\mathrm{H1}^{(1)}
\end{aligned}
\tag{65}
$$

Here, we dropped the superscript for all nested accessor functions to increase the readability. This assertion must imply the postcondition, which is reduced to:

$$\sigma \models m.\mathrm{return}^{(1)} \in \mathrm{CNode\_flip\_strong\_Set} \tag{66}$$

The gap is bridged by the application of the derived fixed-point-induction:

$$
\frac{\bigwedge X.\quad \begin{array}{c}[\sigma \models m.\mathrm{return}^{(1)} \in X] \\ \vdots \\ \sigma \models m.\mathrm{return}^{(1)} \in (\lambda\, \sigma.\, \mathrm{CNode\_flip\_strong\_F}\, \sigma\, X)\end{array}}{\sigma \models m.\mathrm{return}^{(1)} \in (\lambda\, \sigma.\ \mathrm{gfp}(\mathrm{CNode\_flip\_strong\_F}\, \sigma))}
\tag{67}
$$

The example also shows how liberal invariants (a freshly generated object only satisfies such an invariant since the .left and .right attribute are uninitialized) can be used to establish stronger ones (.left and .right always refer to defined objects). In [21] local flags in objects are suggested to switch on and off parts of static class invariants. Our approach does not need such flags (while it can mimic them), rather, we would generate versions of invariants and relate them via co-induction automatically.

8 Conclusion

We presented an extensible universe construction supporting object-oriented data models providing subtyping and (single) inheritance. As syntactic interface for object-oriented data models, we used the annotation language COOL which we interpreted in two logical embeddings called level 1 and level 2. The underlying mapping from object-language types to types in the HOL representation is injective, which implies type-safety. We introduce co-inductive properties on object systems via characteristic sets defined by greatest fixed-points; these sets also give a semantics for class invariants. In our package, constructors and update-operations were handled too.

	Invoice	eBank	Company	Royals and Loyals
number of classes	3	8	7	13
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 2 Importing Different UML/OCL Specifications.

Finally, we integrated COOL inside a Hoare calculus for a conceptual imperative core-language with object creation and object update. We used both interpretations for COOL—level 1 for small-step reasoning, level 2 for big-step reasoning—to verify a program generating a cyclic object graph against its specification.

The universe-construction is supported by a package (developed as part of the HOL-OCL project [12]). Generated theories on object systems can be applied for both object-oriented specification languages like OCL and programming language embeddings using the type-safe shallow embedding technique.

In the context of HOL-OCL, we gained some experimental data that shows the feasibility of the technique: Table 2 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 COOL 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 as well as their complexity.

8.1 Related Work

Datatype packages have been considered mostly in the context of HOL or functional programming languages. Systems like [23,8] build over an S-expression like term universe (co)-inductive sets which are abstracted to (freely generated) datatypes. Paulson's inductive package [32] also uses subsets of the ZF set universe *i*. To the best of our knowledge, this is the first attempt to apply this technology to type-safe object-oriented data models derived on-the-fly from conservative definitions.

There is a substantial body on literature on object-oriented language semantics based on deep embeddings. A more conceptual work is NanoJava [31], which is in spirit quite similar to IMP++, albeit focusing on meta-theoretic proofs like completeness. Although the proofs done with NanoJava are rather smallish (in this respect quite similar to IMP++), it is instructive to compare its rules with the ones of IMP++. We just consider the case of a cast-operation in an own subcalculus on expressions:

$$\frac{A \models_e \{P\} e \left\{ \lambda\, v\, s.\ \begin{pmatrix} \text{case } v \text{ of Null} & \Rightarrow \text{true} \\ \mid \text{Addr} a & \Rightarrow \text{obj } class\, s\, a <_C C \end{pmatrix} \to Q\, v\, s \right\}}{A \models_e \{P\} \text{Cast}\, C\, e \{Q\}}\ (Cast) \tag{68}$$

These side-conditions in subcalculi deciding whether the type of a reference is conform to another type is just superfluous in IMP++, where type-casts were represented as simple equational rules that are amenable to rewriting. The Isabelle/Bali Project [30] followed a similar approach to NanoJava, but for a quite substantial fragment of the Java language. It served as a formal reference semantics in several other projects (see below). Working with this embedding is technically very challenging, both with respect to time and memory consumption. The complexity of side-condition evaluations inspired for some time the development of code-generators for Isabelle. The approach is in principle compatible to open world assumptions, but not easily amenable for modular verification.

The KeY Tool [3,7] is a verification environment integrated into a CASE tool. It offers remarkable support for development and claims a high degree of proof automation. However, it is based on an axiomatic description of Java and C like languages in Dynamic Logic, which is implemented in *taclets*, i. e., purely syntactic transformations of a prover state. For fragments of the substantial rule-sets, formal verifications with respect to an operational Java model in Maude and Isabelle/Bali have been undertaken at some stage of the system development. In contrast, our datatype package makes comprehensive proofs for the consistency of the extensible data model based on an LCF-style kernel allowing only logical manipulations with respect to HOL.

Jive [24] compiles an object-oriented data model into a fixed Isabelle theory and includes this into a derived Hoare calculus over a substantial fragment of Java. The system shares basic ideas with respect to the object model with Spec# (see below). However, the overall construction is based on a closed world assumption and thus, not extensible. An extension of an object-model results in a recompilation that needs at present 20 minutes for small programs.

To the probably most advanced tools belong verification condition generator approaches such as Boogie for Spec# [5,21], which is excellently integrated into a CASE tool. The underlying idea is to compile object-oriented programs into standard imperative ones and to apply a verification condition generator on the latter. The approach requires the generation of a quite substantial axiomatization of an object-oriented memory/machine model, and an explicit first-order representation of object-oriented types within a logical context in which the verification conditions were stated. The second author witnessed several logical inconsistencies in an attempt to verify the memory/machine model of a C variant of the Boogie system with Isabelle. We believe that the properties of our object-oriented memory model, even if taken axiomatically, could provide assurance. If required, our system can generate a proof of consistency for given data models.

Krakatoa [22] and several similar tools for JML follow a similar approach as Boogie for Spec#. While the core, the Why tool supporting an alias-free imperative language, has been verified, the overall tool including object-oriented compilation is not. Inconsistencies of past versions of axiomatic memory models have been reported. The object-oriented model has only a constrained notion of dynamic type and is only partly extensible.

For shallow embeddings, there is the work by Smith et al. [33]. In this approach, however, emphasis is put 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 on class isomorphisms. As the authors admit, this “creates considerable formal overhead.” Subtyping on *objects* must be expressed implicitly via refinement. Somewhat more similar to our work are the encodings provided by Huisman [17] and its follow-up by Jacobs [19]. The approach is based on a straightforward compilation of class systems to families of records; this is in contrast to our work restricted to closed world data models. The follow-up paper, developed in the context of the LOOP tool, states our work on automated class invariant derivation explicitly as desirable future work. The approach by Yatake [35] is similar to Huisman's and suffers from the same limitations; however, the derivation of the elementary data model rules is very similar to ours.

With respect to extensibility of data structures, the idea of using parametric polymorphism is partly folklore in HOL research communities; for example, extensible records and their application for some form of subtyping has been described in HOOL [26]. Since only α-extensions are used, this results in a restricted form of class types with no cast mechanism to α `Object`.

8.2 Future Work

We see the following lines of future research:

Towards a Generic Package. The supported type language as well as the syntax for the co-induction schemes is fixed in our package so far. More generic support for annotation languages like OCL, COOL or its strict version is required to make our package more widely applicable.

Support for Inductive Constraints. By introducing measure-functions over object structures, inductive datatypes can be characterized for defined measures of an object. This paves the way for the usual structural induction and well-founded recursion schemes.

Support of built-in Co-recursion. Co-recursion can be used to define e. g., deep object equalities.

Deriving VCG. Similar to the IMP-theory, verification condition generators for IMP++ programs can be proven sound and complete. This leads to effective program verification techniques based entirely on derived rules.

Acknowledgements We thank Lukas Brügger for valuable discussions and his proof-work within IMP++.

References

1. UML 2.0 OCL specification (2003). Available as OMG document ptc/03-10-14
2. Unified modeling language specification (version 1.5) (2003). Available as OMG document formal/03-03-01
3. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool **4**(1), 32–54 (2005). 10.1007/s10270-004-0058-x
4. Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof, 2nd edn. (2002)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: G. Barthe, L. Burdy, M. Huisman, J.L. Lanet, T. Muntean (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), vol. 3362, pp. 49–69 (2005). 10.1007/b105030
6. Basin, D.A., Kuruma, H., Takaragi, K., Wolff, B.: Verification of a signature architecture with HOL-Z. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) FM 2005: Formal Methods, vol. 3582, pp. 269–285 (2005). 10.1007/11526841_19
7. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, vol. 4334 (2007). 10.1007/978-3-540-69061-0

8. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—lessons learned in formal-logic engineering. In: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (eds.) Theorem Proving in Higher Order Logics (TPHOLS), vol. 1690, pp. 19–36 (1999). 10.1007/3-540-48256-3_3
9. Böhme, S., Leino, K.R.M., Wolff, B.: HOL-Boogie—an interactive prover for the Boogie program-verifier. In: O.A. Mohamed, C. Muñoz, S. Tahar (eds.) Theorem Proving in Higher Order Logics, vol. 5170, pp. 150–166 (2008). 10.1007/978-3-540-71067-7_15
10. Brucker, A.D.: An interactive proof environment for object-oriented specifications. Ph.D. thesis, ETH Zurich (2007). URL `http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007`. ETH Dissertation No. 17097.
11. Brucker, A.D., Rittinger, F., Wolff, B.: HOL-Z 2.0: A proof environment for Z-specifications. Journal of Universal Computer Science **9**(2), 152–172 (2003).
12. Brucker, A.D., Wolff, B.: The HOL-OCL book. Tech. Rep. 525, ETH Zurich (2006). URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006`
13. Brucker, A.D., Wolff, B.: HOL-OCL – A Formal Proof Environment for UML/OCL. In: J. Fiadeiro, P. Inverardi (eds.) Fundamental Approaches to Software Engineering (FASE08), 4961, pp. 97–100 (2008). 10.1007/978-3-540-78743-3_8.
14. Brucker, A.D., Wolff, B.: Extensible universes for object-oriented data models. In: J. Vitek (ed.) ECOOP 2008 – Object-Oriented Programming, 5142, pp. 438–462 (2008). 10.1007/978-3-540-70592-5_19.
15. Drossopoulou, S., Eisenbach, S.: Describing the semantics of Java and proving type soundness. In: J. Alves-Foss (ed.) Formal Syntax and Semantics of Java, vol. 1523, pp. 41–82 (1999). 10.1007/3-540-48737-9_2
16. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. In: J. Alves-Foss (ed.) Formal Syntax and Semantics of Java, vol. 1523, pp. 241–269 (1999). 10.1007/3-540-48737-9_7
17. Huisman, M., Jacobs, B.: Inheritance in higher order logic: Modeling and reasoning. In: M. Aagaard, J. Harrison (eds.) Theorem Proving in Higher Order Logics (TPHOLS), vol. 1869, pp. 301–319 (2000). 10.1007/3-540-44659-1_19
18. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ **23**(3), 396–450 (2001). 10.1145/503502.503505
19. Jacobs, B., Poll, E.: Java program verification at Nijmegen: Developments and perspective. In: K. Futatsugi, F. Mizoguchi, N. Yonezaki (eds.) Software Security—Theories and Systems (ISSS), vol. 3233, pp. 134–153 (2004). 10.1007/b102118
20. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: H. Kilov, B. Rumpe, I. Simmonds (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188 (1999)
21. Leino, K.R.M., Müller, P.: Modular verification of static class invariants. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) FM 2005: Formal Methods, vol. 3582, pp. 26–42 (2005). 10.1007/11526841_4
22. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In: J. Hurd, T.F. Melham (eds.) Theorem Proving in Higher Order Logics (TPHOLS), vol. 3603, pp. 179–194 (2005). 10.1007/11541868_12
23. Melham, T.F.: A package for inductive relation definitions in HOL. In: M. Archer, J.J. Joyce, K.N. Levitt, P.J. Windley (eds.) International Workshop on the HOL Theorem Proving System and its Applications (TPHOLS), pp. 350–357 (1992)
24. Meyer, J., Poetzsch-Heffter, A.: An architecture for interactive program provers. In: S. Graf, M.I. Schwartzbach (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 1785, pp. 63–77 (2000)
25. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF **9**(2), 191–223 (1999). 10.1017/S095679689900341X
26. Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in higher-order logic. In: J. Grundy, M.C. Newey (eds.) Theorem Proving in Higher Order Logics (TPHOLS), vol. 1479, pp. 349–366 (1998). 10.1007/BFb0055146
27. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook **10**(2), 171–186 (1998). 10.1007/s001650050009
28. Nipkow, T., von Oheimb, D.: Java$_{\ell ight}$ is type-safe—definitely. In: ACM Symp. Principles of Programming Languages (POPL), pp. 161–170 (1998). 10.1145/268946.268960
29. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, vol. 2283 (2002). 10.1007/3-540-45949-9

30. von Oheimb, D.: Analyzing Java in Isabelle/HOL: Formalization, type safety and Hoare logic. Ph.D. thesis, Technische Universität München (2001)
31. von Oheimb, D., Nipkow, T.: Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In: L.H. Eriksson, P.A. Lindsay (eds.) FME 2002: Formal Methods—Getting IT Right, vol. 2391, pp. 89–105 (2002). 10.1007/3-540-45614-7_6
32. Paulson, L.C.: A fixedpoint approach to (co)inductive and (co)datatype definitions. In: G. Plotkin, C. Stirling, M. Tofte (eds.) Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 187–211 (2000)
33. Smith, G., Kammüller, F., Santen, T.: Encoding Object-Z in Isabelle/HOL. In: D. Bert, J.P. Bowen, M.C. Henson, K. Robinson (eds.) ZB 2002: Formal Specification and Development in Z and B, vol. 2272, pp. 82–99 (2002). 10.1007/3-540-45648-1_5
34. Winskel, G.: The Formal Semantics of Programming Languages (1993)
35. Yatake, K., Aoki, T., Katayama, T.: Implementing application-specific object-oriented theories in HOL. In: D.V. Hung, M. Wirsing (eds.) Theoretical Aspects of Computing—ICTAC 2005, vol. 3722, pp. 501–516 (2005). 10.1007/11560647_33