An Extensible Encoding of Object-oriented Data Models in HOL
with an Application to IMP++

Achim D. Brucker[1] and Burkhart Wolff[2]

[1]SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany,
[2]Universität des Saarlandes, 66041 Saarbrücken, Germany

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 dynamic and static 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

1. Introduction

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

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

1. *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# [BLS05, LM05], Krakatoa [MPM05] and several similar tools based on the Java Modeling Language (JML) [LBR99]. 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 memory/machine 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 [BLW08].

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., [DE99, FKF99, NvO98, vON02]). In a deep embedding of a language semantics, syntax and types are represented by free datatypes. As a consequence, derived calculi inherit a heavy syntactic bias in form of side-conditions over binding and typing issues. This is unavoidable if one is interested in meta-theoretic properties such as type-safety; however, when reasoning 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) allowing reasoning over Haskell-like programs [MNvOS99] or for HOL-Z [BRW03, BKTW05].

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. In a shallow embedding, an expression E of type T in some object-oriented language must be translated into some HOL-expression E of HOL-type T. Moreover, this translation should preserve well-typedness in both ways. However, an injective mapping does not mean a simple one-to-one conversion; rather, the translation might use a pre-processing making, for example, implicit casts between subtypes and supertypes explicit. Still, we need a formal data model that has support for such semantic properties, like type casting without losing information, built-in.

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.

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, 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++.

In Section 2 we introduce into our meta-language higher-order logic (HOL) and into a syntactic interface for accessors, casts, tests, etc, in 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 [NPW02] is a generic, LCF-style theorem prover implemented in SML. For our object-oriented datatype package, we use the possibility to build 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) [And02] 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 $\tau_1 \Rightarrow (\dots \Rightarrow (\tau_n \Rightarrow \tau_{n+1}) \dots)$ are also written as $[\tau_1, \dots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow$ 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 α set can be defined isomorphic to functions of type $\alpha \Rightarrow$ bool; the element-of-relation $\_ \in \_$ has the type $[\alpha, \alpha\, \text{set}] \Rightarrow$ bool and corresponds basically to the application; in contrast, the set comprehension $\{\_|\_\}$ has type $[\alpha\, \text{set}, \alpha \Rightarrow \text{bool}] \Rightarrow \alpha$ set and corresponds to the λ-abstraction.

In fact, Isabelle provides a two tier type system similar to Haskell: types are annotated with *typeclasses*. Throughout this paper, we assume a type class $\alpha :: \text{bot}$ for all types α that provide an exceptional element $\bot$; for each type in this class a test for definedness is available via def $x \equiv (x \neq \bot)$. The HOL type constructor $\tau_\bot$ assigns to each type τ a type *lifted* by $\bot$. Thus, each type $\alpha_\bot$ is member of the class bot. The function $\lfloor \_ \rfloor : \alpha \Rightarrow \alpha_\bot$ denotes the injection, the function $\lceil \_ \rceil : \alpha_\bot \Rightarrow \alpha$ its inverse for defined values. Based on these definitions, partial functions, i. e., $\alpha \rightharpoonup \beta$, can be represented as total functions of type $\alpha \Rightarrow \beta_\bot$. We define $dom\, f$, called the *domain* of a partial function f, by the set of all arguments of f that do not yield $\bot$.

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:

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

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*; 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:

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

While the latter lines sketch a conventional expression language, the first line contains key elements of object-orientation: the result of path expression may denote (sets of, sequences of) objects, paths can be tested if they are *defined* (∂P) within a store, the *dynamic type* $(P.\texttt{isType}\,(T))$ and the *kind* $(P.\texttt{isKind}\,(T))$ 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 {CNode $\mapsto$ {color $\mapsto$ Boolean}, Node $\mapsto$ {left $\mapsto$ Node, ...}}. Note, however, that the COOL assertions are based on a two-valued semantics (similar to Spec# or JML), whereas OCL [Obj03a] (a constraint language for UML [Obj03b]) 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 [BW08, BW06a], it can be used for an arbitrary assertion language with the techniques presented here. During loading a specification, 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 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

```
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.

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.

3.1. A Typed Objects 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. If we aim towards a conservative formalization of our object store, we cannot model the set T of COOL types directly if we want 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). Thus, 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. 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 $A.\text{left}^{(1)}$ (the superscript denotes that this is an attribute accessor on level 1) for a given state; an access failure will be reported by $\bot$.

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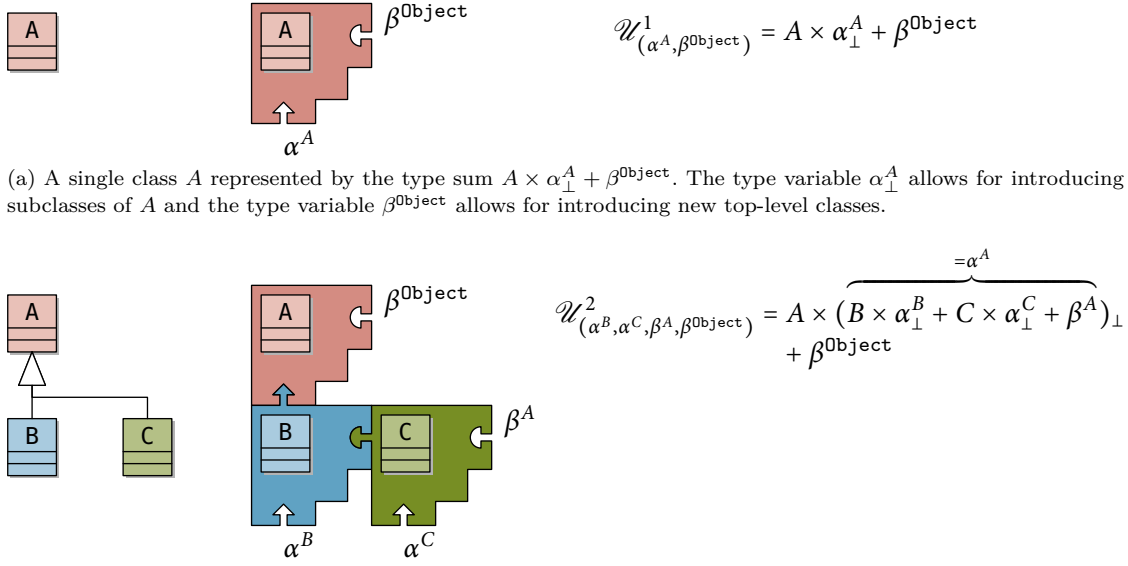
(a) A single class A represented by the type sum $A \times \alpha_\perp^A + \beta^{\texttt{Object}}$. The type variable $\alpha_\perp^A$ allows for introducing subclasses of A and the type variable $\beta^{\texttt{Object}}$ allows for introducing new top-level classes.

(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 $B \times \alpha_\perp^B + C \times \alpha_\perp^C + \beta^A$. 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 $\mathscr{U}^2_{(\alpha^B, \alpha^C, \beta^A, \beta^{\texttt{Object}})}$.

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* $C_t \in \mathfrak{T}$ 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 assign an abstract datatype Node_t with the only element 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 $(t \times \text{unit}) \in \mathfrak{B}$, 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 $\text{Node}_t \times \texttt{Integer} \times \text{oid} \times \text{oid}$ and of class CNode is $\text{CNode}_t \times$ 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 (Object Constraint Language), this is OclAny.

Definition 4 (Object). Let $\text{Object}_t \in \mathfrak{T}$ be the tag of the common supertype Object and oid the type of the object identifiers. We define $\alpha \; \texttt{Object} := \big((\text{Object}_t \times \text{oid}) \times \alpha_\perp\big)$.

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. $\mathscr{U}_\alpha^0 \in \mathfrak{U}$ is the initial universe type with one type variable (hole) α.
2. if $\mathscr{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)} \in \mathfrak{U}$, $n, m \in \mathbb{N}$, $i \in \{0,\ldots,n\}$ and $c \in \mathfrak{B}$ then

   $$\mathscr{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)}\Big[\alpha_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}$$

   This definition covers the introduction of "direct object extensions" by instantiating α-variables.
3. if $\mathscr{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)} \in \mathfrak{U}$, $n, m \in \mathbb{N}$, $i \in \{0,\ldots,m\}$, and $c \in \mathfrak{B}$ then

   $$\mathscr{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)}\Big[\beta_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}$$

   This definition covers the introduction of "alternative object extensions" by instantiating β-variables.

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

$$\mathscr{U}_{(\alpha_C,\beta_C,\beta_N)}^1 = \Big((\text{Node}_t \times \texttt{Integer} \times \text{oid} \times \text{oid})$$
$$\times \big((\text{CNode}_t \times \texttt{Boolean}) \times (\alpha_C)_\perp + \beta_C\big)_\perp + \beta_N\Big)\ \texttt{Object} + Values. \quad (5)$$

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 $\mathscr{U}$ be the universe covering, among others, class C_n, and let $C_0,\ldots,C_{n-1}$ be the supertypes of C, i.e, C_i is inherited from C_{i-1}. The class type of C is defined as:

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

   $$\mathscr{C}_\alpha^0 = \left(C_0 \times \Big(C_1 \times \big(C_2 \times \cdots \times (C_n \times \alpha_\perp)_\perp\big)_\perp\Big)_\perp\right)_\perp \in \mathfrak{C},$$

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

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

$$(\alpha_C, \beta_C)\ \texttt{Node} =$$
$$\Big((\text{Node}_t \times \texttt{Integer} \times \text{oid} \times \text{oid}) \times \big((\text{CNode}_t \times \texttt{Boolean}) \times (\alpha_C)_\perp + \beta_C\big)_\perp\Big)\ \texttt{Object}. \quad (6)$$

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:

$$\text{mk}_{\text{Object}}^{(0)}\ obj = \text{Inl}\ obj \qquad\qquad \text{with type } \alpha\ \texttt{Object} \Rightarrow \mathscr{U}_\alpha^0 \qquad (7)$$

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:

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

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 is a little bit more complex, but follows the same schema: for the injections we have to find the "right" position in the type sum and insert the given object into that position. Further, we define in a similar way projectors for all class attributes. For example, we define the projector for accessing the `left` attribute of the class `Node`:

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

with type (α_1, β) Node $\Rightarrow$ oid$_\perp$ and where base is a variant of snd over lifted tuples:

$$\text{base } x \equiv \begin{cases} b & \text{if } x = \lfloor(a, b)\rfloor, \\ \perp & \text{otherwise.} \end{cases} \tag{10}$$

For attributes with non-value types we return an oid. As we return oid for all class types, we consider this construction not as type-safe in an object-oriented sense. In Section 4, we show how these projectors can be used for defining 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:

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

where OidOf __ accesses the object identifier, __.i$^{(0)}$ and __.right$^{(0)}$ are attribute accessors, and __.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:

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

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 $\perp$:

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

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

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

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
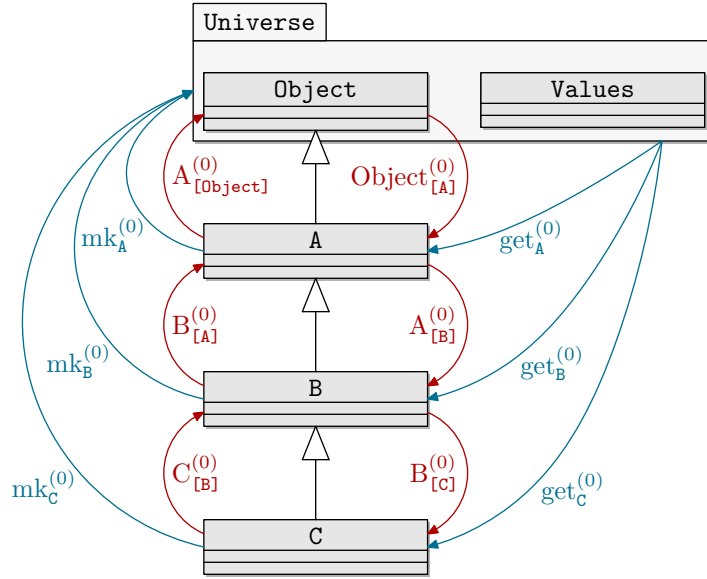
Figure 3. The type casts, e.g., $B_{[C]}$ allow the conversion of a type to its direct successor or predecessor in the type hierarchy. The injections, e.g., mk_B convert a class type to the universe type and the projections, e.g., get_B, convert a universe type to a concrete class type. For a universe 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`).

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.:

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

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

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., $\mathscr{U}^0_\alpha = \alpha\ \mathtt{Object}$, 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 are inverse.

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 $((\alpha_1, \beta)\ \mathtt{Node})\ \mathtt{Set}$) can also contain objects of type `CNode`. For resolving operation overriding, i.e., late-binding, the packages generates operation tables user-defined operations; see [BW06a, Bru07] for details.

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 [Obj03b]). 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:

$$\frac{\text{isUniv}_{\text{A}}^{(0)}\ univ}{\text{isUniv}_{\text{B}}^{(0)}\ univ} \qquad (17.\text{a}) \qquad \frac{\text{isType}_{\text{B}}^{(0)}\ obj}{\text{isKind}_{\text{A}}^{(0)}\ obj} \qquad (17.\text{b})$$

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

$$\frac{\text{isUniv}_{\text{A}}^{(0)}\ univ}{\text{mk}_{\text{A}}^{(0)}(\text{get}_{\text{A}}^{(0)}\ univ) = univ} \qquad (18.\text{a}) \qquad \frac{\text{isType}_{\text{A}}^{(0)}\ obj}{\text{get}_{\text{A}}^{(0)}(\text{mk}_{\text{A}}^{(0)}\ obj) = obj} \qquad (18.\text{b})$$

$$\frac{\text{isType}_{\text{B}}^{(0)}\ obj}{\text{isUniv}_{\text{A}}^{(0)}(\text{mk}_{\text{A}}^{(0)}\ obj)} \qquad (19.\text{a}) \qquad \frac{\text{isUniv}_{\text{A}}^{(0)}\ univ}{\text{isType}_{\text{A}}^{(0)}(\text{get}_{\text{A}}^{(0)}\ univ)} \qquad (19.\text{b})$$

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:

$$\frac{\text{isType}_{\text{A}}^{(0)}\ obj}{obj_{[\text{B}]}^{(0)} = \bot} \qquad (20.\text{a}) \qquad \frac{\text{isType}_{\text{B}}^{(0)}\ obj}{\left((obj_{[\text{A}]}^{(0)})_{[\text{B}]}^{(0)}\right) = obj} \qquad (20.\text{b})$$

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

$$\frac{}{\bot_{[\text{A}]}^{(0)} = \bot} \qquad (21.\text{a}) \qquad \frac{\text{isType}_{\text{C}}^{(0)}\ obj}{(obj_{[\text{B}]}^{(0)})_{[\text{A}]}^{(0)} = obj_{[\text{A}]}^{(0)}} \qquad (21.\text{b})$$

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

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

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

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

As we use a shallow embedding of object-oriented data-structures into HOL, these properties cannot be proven as meta-theoretic property 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:

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

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

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

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:

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

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 $\rightarrow$ Node, is now represented by our construction as follows:

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

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:

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

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

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

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

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 $0,1,2,\ldots$, true,false, $\{\}$ will be represented by constant functions that just drop the state: $\lambda \, \sigma. \, 0, \lambda \, \sigma. \, 1, \lambda \, \sigma. \, 2,\ldots$, $\lambda \, \sigma. \, \text{true}, \lambda \, \sigma. \, \text{false}$, $\lambda \, \sigma. \, \{\}$. 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 expression, i.e., as argument of a COOL operator as in $self. \, \text{i}^{(1)} > 5$, for example, it is implicitly dropped: $\lambda \, \sigma. \, (\lambda \, \sigma. \, \ulcorner(self. \, \text{i}^{(1)}) \sigma \urcorner) \, \sigma > (\lambda \, \sigma. \, 5) \, \sigma$. 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 $V_\alpha(\text{bool}) \Rightarrow V_\alpha(\text{bool}) \Rightarrow V_\alpha(\text{bool})$ if the corresponding operator $\_\_ \wedge \_\_$ of the meta-language HOL has the type $\text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$. 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. [BW06b]) or a strong Kleene-Logic (cf. [BW06a]). Such an alternative allows for a uniform handling of exceptions (like illegal memory access or "1 div 0") as occurring in programming languages like Java. The key difference to the semantics of COOL as presented above is the definition for the operator family:

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

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:

$$\frac{\sigma \; \models \mathrm{isType}_{\mathrm{A}}^{(1)} \; obj}{\sigma \; \models \mathrm{isKind}_{\mathrm{A}}^{(1)} \; obj} \tag{35.a} \qquad\qquad \frac{\sigma \; \models \mathrm{isType}_{\mathrm{C}}^{(1)} \; obj}{(obj_{[\mathrm{B}]}^{(0)})_{[\mathrm{A}]}^{(1)} \; \sigma = obj_{[\mathrm{A}]}^{(1)} \; \sigma} \tag{35.b}$$

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. Subsequently, we will give semantics to the three invariants `positive`, `flip` and `flip_strong`. 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 gfp :: (α set $\Rightarrow$ α set $\Rightarrow$ α set; for technical reasons, we must talk over *characteristic sets* like Node_positive_Set instead of invariants Node_positive, although both are closely connected:

$$\mathrm{Node\_positive}(self) \equiv self \in \mathrm{Node\_positive\_Set} \equiv self \in (\lambda\,\sigma.\; \mathrm{gfp}(\mathrm{Node\_positive\_F}\;\sigma))$$

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`:

$$\mathrm{Node\_positive\_F} :: \mathscr{U}_{(\alpha_{\mathrm{C}},\beta_{\mathrm{C}},\beta_{\mathrm{N}})}^{1} \, \mathrm{St} \Rightarrow \mathscr{U}_{(\alpha_{\mathrm{C}},\beta_{\mathrm{C}},\beta_{\mathrm{N}})}^{1} \, \mathrm{St} \Rightarrow (\alpha_C, \beta_C)\; \texttt{Node} \; \mathrm{set}$$
$$\Rightarrow \mathscr{U}_{(\alpha_{\mathrm{C}},\beta_{\mathrm{C}},\beta_{\mathrm{N}})}^{1} \, \mathrm{St} \Rightarrow (\alpha_C, \beta_C)\; \texttt{Node} \; \mathrm{set}$$

$$\mathrm{Node\_positive\_F} \equiv \lambda\,\sigma.\;\lambda\,X.\;\big\{ self \;\big|\; \sigma \vDash \partial\, self.\mathrm{i}^{(1)} \wedge \sigma \vDash self.\mathrm{i}^{(1)} > 5$$
$$\wedge\; \sigma \vDash \partial\, self.\mathrm{left}^{(1)} \rightarrow \; \sigma \vDash \mathrm{isKind}_{\mathrm{Node}}^{(1)} \; self.\mathrm{left}^{(1)} \tag{36}$$
$$\wedge\; \sigma \vDash (self.\mathrm{left}^{(1)}) \in X$$
$$\wedge\; \sigma \vDash \partial\, self.\mathrm{right}^{(1)} \rightarrow \; \sigma \vDash \mathrm{isKind}_{\mathrm{Node}}^{(1)} \; self.\mathrm{right}^{(1)}$$
$$\wedge\; \sigma \vDash (self.\mathrm{right}^{(1)}) \in X\big\}$$

The first part of this (generated) definition is a straight forward translation of the formula in the **inv** part of the class declaration in Figure 1; the clauses $\sigma \vDash \partial\, self.\mathrm{left}^{(1)} \rightarrow \; \sigma \vDash \mathrm{isKind}_{\mathrm{Node}}^{(1)} \; self.\mathrm{left}^{(1)} \wedge \sigma \vDash (self.\mathrm{left}^{(1)}) \in X$ (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 as the one shown above and

derives the following *class invariant theorem* expressing the intuition of our invariant:

$$\sigma \vDash \text{Node\_positive } \mathit{self} = \ \sigma \vDash \partial \, \mathit{self}.\mathrm{i}^{(1)}$$

$$\wedge \, \sigma \vDash \mathit{self}.\mathrm{i}^{(1)} > 5$$

$$\wedge \ \sigma \vDash \partial \, \mathit{self}.\mathrm{left}^{(1)} \rightarrow \ \sigma \vDash \mathrm{isKind}^{(1)}_{\mathrm{Node}} \, \mathit{self}.\mathrm{left}^{(1)}$$

$$\wedge \, \sigma \vDash \text{Node\_positive } \mathit{self}.\mathrm{left}^{(1)} \qquad\qquad (37)$$

$$\wedge \ \sigma \vDash \partial \, \mathit{self}.\mathrm{right}^{(1)} \rightarrow \ \sigma \vDash \mathrm{isKind}^{(1)}_{\mathrm{Node}} \, \mathit{self}.\mathrm{left}^{(1)}$$

$$\wedge \, \sigma \vDash \text{Node\_positive } \mathit{self}.\mathrm{right}^{(1)}$$

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

$$\text{Node\_positive } \mathit{self} = \ \partial \, \mathit{self}.\mathrm{i}^{(1)} \wedge \mathit{self}.\mathrm{i}^{(1)} > 5$$

$$\wedge \, \partial \, \mathit{self}.\mathrm{left}^{(1)} \rightarrow \ \mathrm{isKind}^{(1)}_{\mathrm{Node}} \, \mathit{self}.\mathrm{left}^{(1)} \wedge \text{Node\_positive } \mathit{self}.\mathrm{left}^{(1)} \qquad (38)$$

$$\wedge \, \partial \, \mathit{self}.\mathrm{right}^{(1)} \rightarrow \ \mathrm{isKind}^{(1)}_{\mathrm{Node}} \, \mathit{self}.\mathrm{right}^{(1)} \wedge \text{Node\_positive } \mathit{self}.\mathrm{right}^{(1)}$$

Since in our package, it is possible to use the kind and type tests inside invariants, one can establishing 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
```
$$\wedge \, \partial \, \mathit{self}.\mathtt{left} \longrightarrow \mathrm{isType}^{(1)}_{\mathrm{Node}} \, \mathit{self}.\mathtt{left}$$
$$\wedge \, \partial \, \mathit{self}.\mathtt{right} \longrightarrow \mathrm{isType}^{(1)}_{\mathrm{Node}} \, \mathit{self}.\mathtt{right}$$

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:

$$\sigma \vDash \text{Node\_positive\_Type\_Set} \subseteq \text{Node\_positive\_Set} \qquad\qquad (39)$$

This kind of theorem is an 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:

$$\text{CNode\_flip\_F} \equiv \lambda \, \sigma. \ \lambda \, X.$$

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

where $\_ {}^{\backprime} \_$ denotes the point-wise application.

Similar to [BW99] or [Pau00] 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 $\mathfrak{C}_N$ and $\mathfrak{K}_N$ be the corresponding type sets of a direct subclass of `N`, then our encoding process proves formally that the characteristic

type set is a subset of the kind set, i.e.:

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

Moreover, that 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:

$$\sigma \vDash self \in \mathfrak{K}_C \longrightarrow \sigma \vDash self_{[\text{Node}]}^{(1)} \in \mathfrak{C}_N \,. \tag{42}$$

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:

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

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

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

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

All other semantic definitions for COOL remain unchanged.

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

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

With these derived rules, reasoning over a set of objects in the store (e.g., as in OCL provided by the construct `::allInstances()`), to establish class invariants 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,

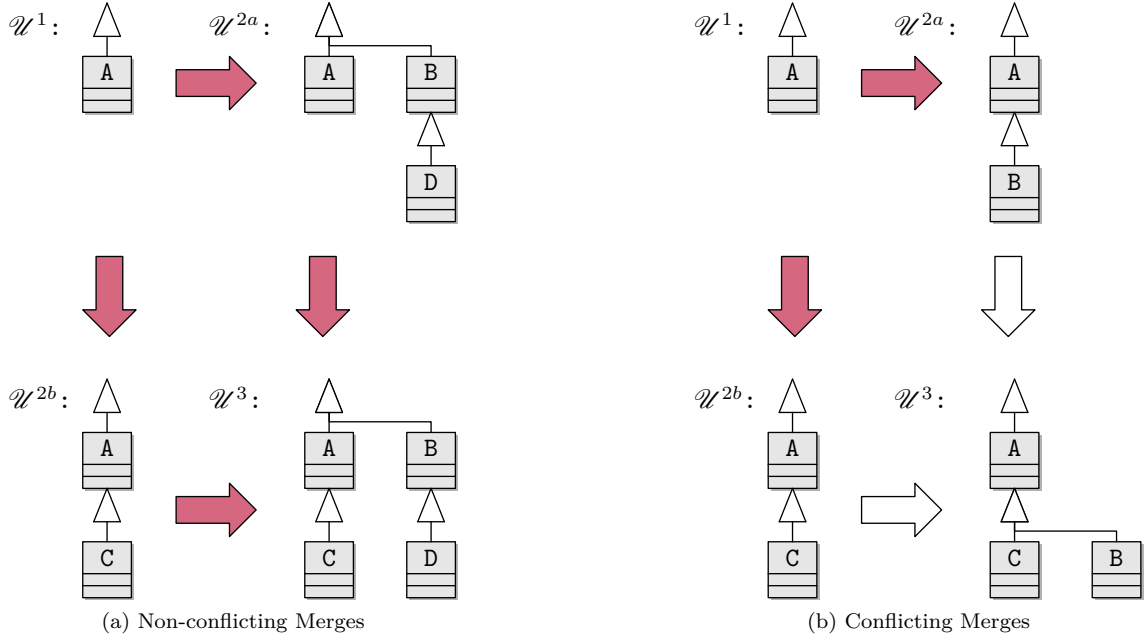(a) Non-conflicting Merges   (b) Conflicting Merges

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.

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 pragmatically more important one; for example, all libraries of the HOL-OCL system [BW06a] 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, 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. Handling Overlapping Merges

Unfortunately, there is a pragmatically 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 "son"-classes into a parent class is relevant since the type sum $\alpha + \beta$ is not a commutative 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,β-"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 an 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
```

Furthermore 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 de-referentiation is in fact defined, we can specify that a specific de-referentiation $self.\mathrm{left}^{(1)}$ has a given dynamic type. An analogous invariant $\mathrm{Inv}_{\mathrm{left}}(self)$ can be defined co-inductively. From this invariant, we can directly infer facts like $\mathrm{isType}_{\mathrm{Node}}^{(1)}\,(self.\mathrm{left}^{(1)})$, and $\mathrm{isType}_{\mathrm{Node}}^{(1)}\,(self.\mathrm{left}^{(1)}.\mathrm{left}^{(1)})$, 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 $\mathrm{Inv}_{\mathrm{left}}(self)$ 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:

$$\mathrm{ObjSetA}_{\mathrm{left}}\ 0\ self\ \sigma = \{\}$$
$$\mathrm{ObjSetA}_{\mathrm{left}}\ k\ self\ \sigma = \mathrm{if}\ \sigma \models \partial\,self\ \mathrm{then}\{\} \tag{47}$$
$$\mathrm{else}\ \{self\} \cup \mathrm{ObjSetA}_{\mathrm{left}}\ (k-1)\ (self.\mathrm{left}^{(1)}\ \sigma)\ \sigma$$

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

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

Moreover, we can add an *state invariant*, on top of 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:

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

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 [IPW01], 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 straight-forward 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. No side-conditions are necessary related to well-formedness of objects, to 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 [vON02].

We will show in a standard exercise that a compact Hoare-Logic can be derived from a denotational semantics for IMP++. As basis, we use IMP [Nip98], a canonical imperative core language available in the Isabelle/HOL library; this language has been inspired by a standard textbook on program semantics [Win93]. 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 individual step of IMP++ can establish it.

In contrast to the previous sections where definitions and proofs were done automatically for all classes and attributes, the proof presented in this section are done 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:

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

SKIP denotes the empty, successfully terminating command, EXN the program that raises the exception (IMP++ possesses only one). The generic command CMD takes as argument a function α cmd which is a synonym for a function $\alpha \, \mathrm{state} \Rightarrow \alpha \, \mathrm{state}_\perp$. Thus, a α 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 α bexp which is a synonym for $\alpha \, \mathrm{state} \Rightarrow \mathrm{bool}$ (resp. $\alpha \, \mathrm{state} \Rightarrow \mathrm{bool}_\perp$ in the case of a strict version of COOL). Any COOL expression has a type which is an instance of α bexp, thus, it can be also used as control expression in IMP++. In the case of a strict version of COOL, as used in [BW06b], the type α bexp is defined as synonym to $\alpha \, \mathrm{state} \Rightarrow \mathrm{bool}_\perp$.

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 $(\alpha :: \mathrm{bot} \, \mathrm{state}_\perp \times \alpha \, \mathrm{state}_\perp) set$. As a consequence, we need as prerequisite the "strict extension" $\_ \circ_\perp \_$ of type $(\beta_\perp \times \gamma_\perp) \, \mathrm{set} \Rightarrow (\alpha_\perp \times \beta_\perp) \, \mathrm{set} \Rightarrow (\alpha_\perp \times \gamma_\perp) \, \mathrm{set}$ on relations:

$$r \circ_\perp s \equiv \{(\perp, \perp)\} \cup \{(x, z). \; \mathrm{def} \; x \wedge (\exists y. \; \mathrm{def} \; y \wedge (x, y) \in s \wedge (y, z) \in r)\}$$
$$\cup \{(x, z). \; \mathrm{def} \; x \wedge (\exists y. \neg \, \mathrm{def} \; y \wedge (x, y) \in s \wedge z = \perp)\} \quad (51)$$

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

$$\frac{}{\vDash \{\odot\, P\}\, \mathtt{SKIP}\{\odot\, P\}} \tag{55.b}$$

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

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

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

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

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

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

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

$$C(\mathtt{SKIP}) = \mathrm{Id} \tag{52a}$$

$$C(\mathtt{EXN}) = \{(s,t).\, t = \bot\} \tag{52b}$$

$$C(\mathtt{CMD}\, f) = \{(s,t).\, s = \bot \wedge t = \bot\} \cup \{(s,t).\, \mathrm{def}\, s \wedge t = f\, \ulcorner s\urcorner\} \tag{52c}$$

$$C(c_0;c_1) = C(c_1) \circ_\bot C(c_0) \tag{52d}$$

$$C(\mathtt{IF}\, b\, \mathtt{THEN}\, c_1\, \mathtt{ELSE}\, c_2) = \{(s,t).\, s = \bot \wedge t = \bot\}$$
$$\cup \{(s,t).\, \mathrm{def}\, s \wedge b\, \ulcorner s\urcorner = \mathrm{true} \wedge (s,t) \in Cc_1\} \tag{52e}$$
$$\cup \{(s,t).\, \mathrm{def}\, s \wedge b\, \ulcorner s\urcorner = \mathrm{false} \wedge (s,t) \in Cc_2\}$$

$$C(\mathtt{WHILE}\, b\, \mathtt{DO}\, c) = \mathrm{lfp}(\Gamma\, b\, (C\, c)) \tag{52f}$$

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

$$\Gamma\, b\, cd \equiv (\lambda\, \phi.\, \{(s,t).\, s = \bot \wedge t = \bot\} \cup \{(s,t).\, \mathrm{def}\, s \wedge b\, \ulcorner s\urcorner = \mathrm{true} \wedge (s,t) \in (\phi \circ_\bot cd)\}$$
$$\cup \{(s,t).\, \mathrm{def}\, s \wedge b\, \ulcorner s\urcorner = \mathrm{false} \wedge s = t\}) \tag{53}$$

7.3. A Derived Hoare-Logic

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

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

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 $\vDash$ directly, avoiding a detour via a derivability notion $\vdash$. Moreover, we use the abbreviation $\odot\, P$ for $\lambda\, \sigma.\, \mathrm{def}\, \sigma \wedge P\sigma$. Thus, assertions like $\vDash \{\odot\, P'\}c\{\odot\, Q'\}$ relate "non-exception" states allowing inference of normal behavior. The derived calculus is now surprisingly standard (see Table 1).

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 straight-forward:

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

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

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.

$$\mathrm{new_{Node}}\ oid \equiv {}_{\lfloor}((\mathrm{Object_t}, oid), {}_{\lfloor}((\mathrm{Node_t}, \bot, \bot, \bot, \bot)_{\rfloor})_{\rfloor} \tag{61}$$

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

$$self.\,\mathrm{new}^{(1)}_{\mathrm{left}}\ \equiv \lambda\,\sigma.\ \mathrm{let}\ \sigma' = \sigma(\mathrm{newOid}\,\sigma := \mathrm{new_{Node}}\ (\mathrm{newOid}\,\sigma))$$
$$\mathrm{in}\ self.\,\mathrm{set}^{(1)}_{\mathrm{left}}\ (\mathrm{new_{Node}}\ (\mathrm{newOid}\,\sigma))\sigma' \tag{62}$$

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

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

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

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

7.5. An Example in IMP++.

A program that produces the smallest non-trivial Object system satisfying the CNode invariant looks in a fictive object-oriented language as follows:

```
method Node m();
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
```

Well, the method as such cannot be represented in IMP++ because we did not provide syntax for that. 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 in IMP++.

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

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

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 as Isabelle as proof goal follows,:

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

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

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

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

$$\sigma \vDash m.\,\text{return}^{(1)} \in \text{CNode\_flip\_strong\_Set} \tag{66}$$

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

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

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 [LM05] 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.

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 [BW06a]).

	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.

Generated theories on object systems can be applied for object-oriented specification languages as OCL as well as programming language embeddings using the type-safe shallow 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 [Mel92, BW99] build over an S-expression like term universe (co)-inductive sets which are abstracted to (freely generated) datatypes. Paulsons inductive package [Pau00] 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[vON02], 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:

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

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 [vO01] 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 [ABB+05, BHS07] is a verification environment integrated into a CASE tool. It offers remarkable support for development and claims for 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 [MPH00] 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# [BLS05, LM05], 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 [MPM05] and several similar tools for JML follow a similar approach than 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 [SKS02]. 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 admits, this "creates considerable formal overhead." Subtyping on *objects* must be expressed implicitly via refinement. Somewhat more similar to our work are the encoding provided by Huismann [HJ00] and its follow-up by Jacobs [JP04]. The approach is based on a straight-forward 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 acsloop tool, states our work on automated class invariant derivation explicitly as desirable future work. The approach by Yatake [YAK05] is similar to Huismanns 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 [NW98]. 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.

Acknowledgments

References

[ABB$^+$05]   Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.

[And02]   Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002.

[BHS07]   Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2007.

[BKTW05] David A. Basin, Hironobu Kuruma, Kazuo Takaragi, and Burkhart Wolff. Verification of a signature architecture with HOL-Z. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 269–285, Heidelberg, 2005. Springer-Verlag.

[BLS05]   Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Heidelberg, 2005. Springer-Verlag.

[BLW08]   Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL-Boogie—an interactive prover for the Boogie program-verifier. In César Muñoz, Sophiène Tahar, and Otmane Ait Mohamed, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2008. To appear.

[Bru07]   Achim D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. Ph.d. thesis, ETH Zurich, March 2007. ETH Dissertation No. 17097.

[BRW03]   Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.

[BW99]   Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL—lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36, Heidelberg, 1999. Springer-Verlag.

[BW06a]   Achim D. Brucker and Burkhart Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[BW06b]   Achim D. Brucker and Burkhart Wolff. A package for extensible object-oriented data models with an application to IMP++. In Serge Autexier and Heiko Mantel, editors, *Verification Workshop (VERIFY)*, 2006.

[BW08]   Achim D. Brucker and Burkhart Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In José Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, 2008.

[DE99]   Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java and proving type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82, Heidelberg, 1999. Springer-Verlag.

[DR07]   Ádám Darvas and Nicole Rauch. On Jive. Private e-mail communication., 2007.

[FKF99]   Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269, Heidelberg, 1999. Springer-Verlag.

[HJ00]   Marieke Huisman and Bart Jacobs. Inheritance in higher order logic: Modeling and reasoning. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 301–319, 2000.

[IPW01]   Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[JP04]   Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security—Theories and Systems (ISSS)*, volume 3233 of *Lecture Notes in Computer Science*, pages 134–153, Heidelberg, 2004. Springer-Verlag.

[LBR99]   Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Dordrecht, 1999.

[LM05]   K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42, Heidelberg, 2005. Springer-Verlag.

[Mel92]   Thomas F. Melham. A package for inductive relation definitions in HOL. In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *International Workshop on the HOL Theorem Proving System and its Applications (TPHOLs)*, pages 350–357, Los Alamitos, CA, USA, 1992. IEEE Computer Society.

[MNvOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

[MPH00]   Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 63–77, Heidelberg, 2000. Springer-Verlag.

[MPM05]   Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194, Heidelberg, 2005. Springer-Verlag.

[Nip98]   Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[NPW02]   Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[NvO98]   Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe—definitely. In *ACM Symp. Principles of Programming Languages (POPL)*, pages 161–170, New York, NY USA, 1998. ACM Press.

[NW98]   Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcolm C. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1479 of *Lecture Notes in Computer Science*, pages 349–366, Heidelberg, 1998. Springer-Verlag.

[Obj03a] UML 2.0 OCL specification, October 2003. Available as OMG document ptc/03-10-14.

[Obj03b]   Unified modeling language specification (version 1.5), March 2003. Available as OMG document formal/03-03-01.

[Pau00]   Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 187–211. MIT Press, Cambridge, Massachusetts, 2000.

[SKS02]   Graeme Smith, Florian Kammüller, and Thomas Santen. Encoding Object-Z in Isabelle/HOL. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 82–99, Heidelberg, 2002. Springer-Verlag.

[vO01]   David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.

[vON02]   David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME 2002: Formal Methods— Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105, Heidelberg, 2002. Springer-Verlag.

[Win93]   Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.

[YAK05]   Kenro Yatake, Toshiaki Aoki, and Takuya Katayama. Implementing application-specific object-oriented theories in HOL. In Dang Van Hung and Martin Wirsing, editors, *Theoretical Aspects of Computing—ICTAC 2005*, volume 3722 of *Lecture Notes in Computer Science*, pages 501–516, Heidelberg, 2005. Springer-Verlag.