Extensible Universes for Object-oriented Data Models

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

[1] SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com
[2] Universität des Saarlandes, 66041 Saarbrücken, Germany
wolff@wjpserver.cs.uni-sb.de

Abstract We present a datatype package that enables the shallow embedding technique to object-oriented specification and programming languages. This datatype package incrementally compiles an object-oriented data model to a theory containing object-universes, constructors, accessors functions, coercions between dynamic and static types, characteristic sets, their relations reflecting inheritance, and the necessary class invariants. The package is conservative, i. e., all properties are derived entirely from axiomatic definitions. As an application, we use the package for an object-oriented core-language called IMP++, for which correctness of a Hoare-Logic with respect to an operational semantics is proven.

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 logic or higher-order logic (HOL). For this reason, there is a tangible conceptual gap between the verification of functional and imperative programs on the one hand and imperative 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) *pre-compilation* into standard logic, and 2) *deep-embeddings* into a meta-logic. As pre-compilation tools, for example, we consider Boogie for Spec# [3,15] and tools based on the Java Modeling Language (JML) such as Krakatoa [16]. The underlying idea is to compile object-oriented programs into standard imperative ones and to apply a verification condition generator on the latter. While technically sometimes very advanced, the foundation of these tools is quite problematic: The compilation in itself is not verified, and it is not clear if the generated conditions are sound with respect to the (usually complex) operational semantics. Among the tools based on deep-embeddings, there is a sizable body of literature on formal models of Java-like languages (e. g., [6,11,12,21,23]). In a deep embedding of a language semantics, syntax and types are represented by free datatypes. As a consequence, derived

Jan Vitek (Ed.): ECOOP 2008, LNCS 5142, pp. 438–462, 2008.

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 proofs for type-safety, soundness of Hoare-Calculi and even soundness of verification condition generators are done, none of the mentioned deep embeddings has been used for substantial proof work in applications.

In contrast, the *shallow embedding* technique has been used for semantic representations such as HOL itself (in Isabelle/Pure), for HOLCF (in Isabelle/HOL) allowing reasoning over Haskell-like programs [18] or, for HOL-Z [8]. These embeddings have been used for substantial applications (e. g., [4]). The essence of a shallow embedding is to represent object-language binding and typing directly in the binding and typing machinery of the meta-language. Thus, many side-conditions are simply unnecessary; type-safety, for example, has been proven implicitly when deriving computational rules from semantic definitions. Since implicit side-conditions are "implemented" by built-in mechanisms, they are handled orders of magnitude faster compared to an explicit treatment.

At first sight, it seems impossible to apply the shallow embedding technique to object-oriented languages in HOL. In this technique, an expression `E` of type `T` in some object-oriented language must be translated into some HOL-expression E of HOL-type T. The translation should preserve well-typedness in both ways. However, by "translation" we do not mean a simple one-to-one conversion; rather, the translation might use the object-oriented type system for a pre-processing step, making, for example, implicit coercions between subtypes and supertypes explicit. Still, this requires a representation where subtyping is embedded into parametric polymorphism.

The type representation problem becomes apparent when defining the most fundamental concept of an object-oriented language, namely its underlying *state* called *object structure*. *Objects* are abstract representations of pieces of memory that are 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 viewed as the type of the class they are belonging to. Obviously, object structures are maps of type $\text{oid} \Rightarrow \mathscr{U}$ 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 untyped or "too large" for (simply) typed HOL, where all type sums must be finite), one could think of generating an object universe only for each given system of classes. Ignoring subtyping and inheritance for a moment, this would result in a universe $\mathscr{U}^0 = A + B$ for some class system with the classes A and B. Unfortunately, such a construction is not extensible: If we add a new class to an existing class system, say D, then the "obvious" construction $\mathscr{U}^1 = A+B+D$ results in a *different* type to $\mathscr{U}^0$, 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 systems, which makes it clearly unfeasible.

As contributions of this paper, we will present a novel universe construction which represents subtyping within parametric polymorphism in a preserving manner *and* which is extensible. This construction is used in a novel kind of datatype-package (implemented for Isabelle/HOL), i. e., a kind of logic compiler that generates for each class system and its extensions (for example, given as class models in a standardized format like XMI) various conservative definitions representing an object-oriented data theory. This includes the definition of constructors and accessors, coercions between types, tests, characteristic sets of objects. On this basis, properties reflecting subtyping and proof principles like class invariants are automatically derived. Further, we apply this datatype-package for a small imperative language with object-oriented features and show the soundness of a Hoare-Calculus.

2 Formal and Technical Background

Isabelle [22] is a generic, LCF-style theorem prover implemented in SML. For our objects-oriented datatype package, we use the possibility to build SML programs performing symbolic computations over HOL formulae in a logically safe way. Isabelle/HOL offers support for checks for conservatism of definitions, datatypes, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers.

Higher-order logic (HOL) [2] is a classical logic with equality enriched by total polymorphic higher-order functions. It is more expressive than first-order logic, e. g., induction schemes can be expressed inside the logic. HOL is based on the typed λ-calculus, i. e., the *terms* of HOL are λ-expressions. The *application* is written by juxtaposition $E\,E'$, and the *abstraction* is written $\lambda\, x.\, E$. Types may be built from *type variables* (like α, β, optionally annotated by *type classes*, e. g., $\alpha :: order$) or *type constructors* (e. g., bool). Type constructors may have arguments (e. g., α list). The type constructor for the function space is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ are also written as $[\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow$ bool, where bool is the fundamental logical type. The logical connectives $\_ \wedge \_$, $\_ \vee \_$, $\_ \rightarrow \_$ of HOL have type $[\text{bool}, \text{bool}] \Rightarrow$ bool, $\neg\_$ has type bool $\Rightarrow$ bool. The quantifiers $\forall \_.\, \_$ and $\exists \_.\, \_$ have type $[\alpha\ \text{set}, \alpha \Rightarrow \text{bool}] \Rightarrow$ bool. Quantifiers may range over higher order types, i. e., functions.

The type discipline rules out paradoxes such as Russels 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.

Isabelle supports conservative theory extensions schemes; this means that a theory (viewed as a pair of a signature Σ and a set of axioms Φ) can only by extended by type-declarations, constant-declaration and axioms with a particular form. For example, the conservative extensions of a *constant definition*

is constrained to a constant declaration $c :: \tau$ and an axiom $c = E$ where c is fresh, i. e., not contained in the previous signature, E does neither contain free (type) variables nor c (these syntactic conditions are checked by Isabelle). For conservative extension schemes such as constant definitions, the extended theory is consistent ("has models") if the original theory is consistent [13].

For our work, 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 $\text{def}\, x \equiv (x \neq \bot)$. 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 $\llcorner\_\lrcorner : \alpha \to \alpha_\bot$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\bot \to \alpha$ its inverse for defined values.

3 Typed Object Universes in an Object Store

In this section, we focus on the map associating an expression `E` of type `T` to a HOL expression E of type T. The cornerstones of this map are the (functional) constructors,[3] selectors, tests for dynamic type and kind as well as cast operations between objects along the class hierarchy.

As a pre-requisite, we have to define the families $\mathscr{U}^i$ of object universes. Each $\mathscr{U}^i$ comprises all *value types* and an extensible *class type representation* induced by a class hierarchy. To each class, a *class type* (like `Node` or `Object`) is associated which represents the set of *object instances* or *objects*. The extensibility of a universe type is reflected by "holes" (polymorphic variables), that can be filled when "adding" extensions to a class. 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 over $\mathscr{U}^i$ remain valid with respect to $\mathscr{U}^{i+1}$, see Figure 1 for an illustration of the main ideas of the construction we present in the following.

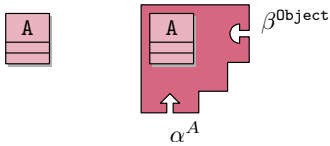
A Formal Framework of Object Structure Encodings. We will present the framework of our object encoding together with a small example: assume a class `Node` with an attribute `i` of type integer and two attributes `left` and `right` of type `Node`, and an inherited class `Cnode` (i. e., `Cnode` is a subtype of `Node`) with an attribute `color` of type `Boolean`.

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. For the class attributes we define:

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

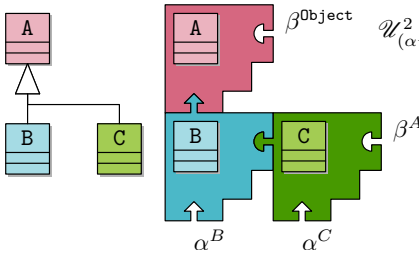
1. $\{\texttt{Boolean}, \texttt{Integer}, \texttt{Real}, \texttt{String}, \text{oid}\} \subset \mathfrak{A}$, *and*
2. $\{a\ \texttt{Set}, a\ \texttt{Sequence}, a\ \texttt{Bag}\} \subset \mathfrak{A}$ *for all* $a \in \mathfrak{A}$.

[3] These constructors only create a value, in contrast to constructors in object-oriented languages that additionally bind this value to a fresh oid in the memory.



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

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



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

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

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

In principle, classes are Cartesian products of its attribute types extended by an abstract type ensuring uniqueness.

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

Tag types allow for building a strongly typed universe (with regard to the object-oriented type system), e. g., for class `Node` we assign an abstract datatype Node_t with the only element Node_key. 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 a special HOL type denoting the empty product.*
2. *if* $t \in \mathfrak{T}$ *is a tag type and* $a_i \in \mathfrak{A}$ *for* $i \in \{0, \ldots, n\}$ *then* $(t \times a_0 \times \cdots \times a_n) \in \mathfrak{B}$.

Thus, the base object type of class `Node` is $\text{Node}_\text{t} \times$ `Integer` $\times$ oid $\times$ oid and of class `Cnode` is $\text{Cnode}_\text{t} \times$ `Boolean`.

Without loss of generality, we assume in our object model a common supertype of all objects. In the case of OCL, this is `OclAny`, in the case of Java this is `Object`. This assumption is no restriction because such a common supertype can always be added to a given class structure.

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,*

1. *in the* non-referential *setting, we define* $\alpha\,\texttt{Object} := (\text{Object}_t \times \alpha_\perp)$.
2. *in the* referential *setting, we define* $\alpha\,\texttt{Object} := \big((\text{Object}_t \times \text{oid}) \times \alpha_\perp\big)$.

In the referential setting, 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. However, for other object-oriented semantics the non-referential setting is appropriate, where objects are viewed more like values. The consequences of this choice is discussed elsewhere in more detail [9]. 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* $\mathfrak{U}_{ref}$ *resp.* $\mathfrak{U}_{nref}$ *(abbreviated* $\mathfrak{U}_x$*) is inductively defined by:*

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

Here, item 2 covers the special case of introducing the first subtype by instantiating the α-variable and item 3 covers the general case of introducing further subtypes by instantiating the corresponding β-variable.

The initial universe $\mathscr{U}^0_\alpha$ represents the common supertype (i. e., `Object`) of all classes, i. e., a simple definition would be $\mathscr{U}^0_\alpha = \alpha\,\texttt{Object}$. However, we will need the ability to also store value types: *Values* = `Real`+`Integer`+`Boolean`+`String`. Therefore, we define the initial universe type by $\mathscr{U}^0_\alpha = \alpha\ \texttt{Object} + \mathit{Values}$. Continuing our example we extend the initial universe $\mathscr{U}^0_{(\alpha)}$, in parallel, with the classes `Node` and `Cnode`. This extension leads to the following successor universe type:

$$\mathscr{U}^1_{(\alpha_\text{C},\beta_\text{C},\beta_\text{N})} \equiv \Big((\text{Node}_\text{t} \times \texttt{Integer} \times \text{oid} \times \text{oid}) \times \big((\text{Cnode}_\text{t} \times \texttt{Boolean}) \times (\alpha_\text{C})_\perp + \beta_\text{C}\big)_\perp + \beta_\text{N}\Big)\ \texttt{Object} + \mathit{Values}$$

We pick up the idea of a universe representation without values for a class with all its extensions (subtypes). For each class we construct a type that describes this 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*
   $\mathscr{C}^0_\alpha = \Big(C_0 \times \Big(C_1 \times \big(C_2 \times \cdots \times (C_n \times \alpha_\perp)_\perp\big)_\perp\Big)_\perp\Big)_\perp \in \mathfrak{C}$,
2. $\mathfrak{U}_\mathfrak{C} \supset \mathfrak{C}$*, where* $\mathfrak{U}_\mathfrak{C}$ *is the set of universe types with* $\mathscr{U}^0_\alpha = \mathscr{C}^0_\alpha$.

Thus in our example we construct for the class type of class `Node` the type abbreviation:

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

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

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

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

We see our definition as the more general one, since it allows for "partial objects" potentially relevant for other object-oriented semantics for programming languages. For example Java, for which partial class objects may occur during construction. This paves the way for establishing the definedness of an object step by step.

Furthermore, since the injections and projections are only built to define attribute accessors, partial objects are hidden in our language.

In both cases 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. This choice has the consequence that constructor arguments may be undefined.

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

$$\mathrm{mk}_{\mathrm{Object}}\, o \equiv \mathrm{Inl}\, o \qquad \text{with type } \alpha\,\texttt{Object} \to \mathscr{U}^0_\alpha$$

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

$$\mathrm{get}_{\mathrm{Object}}\, u \equiv \begin{cases} k & \text{if } u = \mathrm{Inl}\, k \\ \varepsilon\, k.\, \mathrm{true} & \text{if } u = \mathrm{Inr}\, k \end{cases} \qquad \text{with type } \mathscr{U}^0_\alpha \to \alpha\,\texttt{Object}.$$

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

$$obj.\,\mathrm{left}^{(l)} \equiv (\mathrm{fst} \circ \mathrm{snd} \circ \mathrm{snd} \circ \mathrm{fst})\ulcorner \mathrm{base}\ obj\urcorner$$

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

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

This construction is not yet type-safe. Nevertheless, this can be easily extended to a type-safe one by adding a unique abstract type for each class type (see Section 4 for details).

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. This is done for `Object` via

$$\mathrm{isUniv}_{\mathrm{Object}}\, u \equiv \begin{cases} \mathrm{true} & \text{if } u = \mathrm{Inl}\, k \\ \mathrm{false} & \text{if } u = \mathrm{Inr}\, k \end{cases} \qquad \text{with type } \mathscr{U}_\alpha^0 \to \mathrm{bool}.$$

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, e. g., not equal to $\perp$:

$$\mathrm{isKind}_{\mathrm{Object}}\, o \equiv \mathrm{def}\, o \qquad \text{with type } \alpha\ \texttt{Object} \to \mathrm{bool}.$$

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

$$\mathrm{isType}_{\mathrm{Object}}\, o \equiv \mathrm{isKind}_{\mathrm{Object}} \wedge \neg\big((\mathrm{def} \circ \mathrm{base})\, o\big) \quad \text{with type } \alpha\ \texttt{Object} \to \mathrm{bool}.$$

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

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

$$\mathrm{Node}_{[\mathrm{Object}]} \equiv \mathrm{get}_{\mathrm{Object}} \circ \mathrm{mk}_{\mathrm{Node}} \text{ with type } (\alpha_1, \beta)\ \mathrm{Node} \to (\alpha_1, \beta_1)\ \texttt{Object},$$
$$\mathrm{Object}_{[\mathrm{Node}]} \equiv \mathrm{get}_{\mathrm{Node}} \circ \mathrm{mk}_{\mathrm{Object}} \text{ with type } (\alpha_1, \beta_1)\ \texttt{Object} \to (\alpha_1, \beta_1)\ \mathrm{Node}.$$

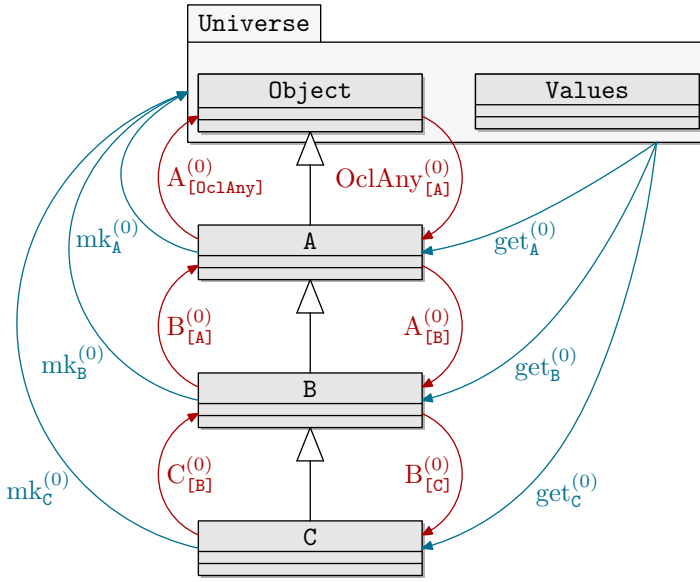


Figure 2. The type casts, e. g., $B_{[C]}$ allow for the conversion of a type to its direct successor or predecessor in the type hierarchy. The injections, e. g., mk_B convert a class type to the universe type and the projections, e. g., get_B, convert a universe type to a concrete class type. For a 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`).

These type-casts are changing the *static type* of an object, while the *dynamic type* remains unchanged. Figure 2 summarizes this construction for the three classes `A`, `B`, and `C`.

Note, for a universe construction without values, e. g., $\mathscr{U}_\alpha^0 = \alpha$ `Object`, the universe type and the class type for the common supertype are the same. In that case there is a particular 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 theorems 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.

3.1 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-oriented (e. g., in the sense of language like Java or Smalltalk or the UML standard [1]). These theorems are proven for each model, e. g., during loading 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 comprising the classes `A`, `B` and `C` where `B` is a subclass of `A` and `C` is a subclass of `B` (recall Figure 2). We start by proving this subtype relation for both our class type and universe type representation:

$$\frac{\mathrm{isUniv}_{\mathrm{A}}^{(0)}\ univ}{\mathrm{isUniv}_{\mathrm{B}}^{(0)}\ univ} \qquad \frac{\mathrm{isType}_{\mathrm{B}}^{(0)}\ obj}{\mathrm{isKind}_{\mathrm{A}}^{(0)}\ obj}$$

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

$$\frac{\mathrm{isUniv}_{\mathrm{A}}^{(0)}\ univ}{\mathrm{mk}_{\mathrm{A}}^{(0)}(\mathrm{get}_{\mathrm{A}}^{(0)}\ univ) = univ} \qquad \frac{\mathrm{isType}_{\mathrm{A}}^{(0)}\ obj}{\mathrm{get}_{\mathrm{A}}^{(0)}(\mathrm{mk}_{\mathrm{A}}^{(0)}\ obj) = obj}$$

$$\frac{\mathrm{isType}_{\mathrm{B}}^{(0)}\ obj}{\mathrm{isUniv}_{\mathrm{A}}^{(0)}(\mathrm{mk}_{\mathrm{A}}^{(0)}\ obj)} \qquad \frac{\mathrm{isUniv}_{\mathrm{A}}^{(0)}\ univ}{\mathrm{isType}_{\mathrm{A}}^{(0)}(\mathrm{get}_{\mathrm{A}}^{(0)}\ univ)}$$

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:

$$\frac{\mathrm{isType}_{\mathrm{A}}^{(0)}\ obj}{obj_{[\mathrm{B}]}^{(0)} = \bot} \qquad \frac{\mathrm{isType}_{\mathrm{B}}^{(0)}\ obj}{\big((obj_{[\mathrm{A}]}^{(0)})_{[\mathrm{B}]}^{(0)}\big) = obj}$$

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

$$\frac{}{\bot_{[\mathrm{A}]}^{(0)} = \bot} \qquad \frac{\mathrm{isType}_{\mathrm{C}}^{(0)}\ obj}{(obj_{[\mathrm{B}]}^{(0)})_{[\mathrm{A}]}^{(0)} = obj_{[\mathrm{A}]}^{(0)}}$$

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

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

$$\overline{(obj.\,\mathrm{set}_{\mathrm{a}}^{(0)}\ x).\,\mathrm{set}_{\mathrm{a}}^{(0)}\ y = obj.\,\mathrm{set}_{\mathrm{a}}^{(0)}\ y}$$

$$\overline{(obj.\,\mathrm{set}_{\mathrm{a}}^{(0)}\ x).\,\mathrm{set}_{\mathrm{b}}^{(0)}\ y = (obj.\,\mathrm{set}_{\mathrm{b}}^{(0)}\ y).\,\mathrm{set}_{\mathrm{a}}^{(0)}\ x}$$

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 models.

4 The Package

The previously described construction is the foundation of the datatype package of HOL-OCL [10,7,9]. For a given class system, described as UML class model, the datatype package of HOL-OCL generates may definitions (the subset of definitions presented in the previous section is marked by $\_ \equiv \_$). Technically, our datatype package supports the standardized XMI format as input (see [9] for implementation details). Besides, it proves automatically several theorems over the imported specification; these theorems are proven for each class, e. g., during loading a specific class model. This includes properties of the object structure, e. g., that our conversion between universe representations and object representation is lossless. This property is characterized by the following two properties, which are, among others, proven automatically by our datatype package:

$$\mathrm{isKind}_{\mathrm{C}}\, o \Longrightarrow \mathrm{get}_{\mathrm{C}}(\mathrm{mk}_{\mathrm{C}}\, o) = o \quad \text{and} \quad \mathrm{isUniv}_{\mathrm{C}}\, u \Longrightarrow \mathrm{mk}_{\mathrm{C}}(\mathrm{get}_{\mathrm{C}}\, u) = u\,.$$

Our construction works also for the encoding of *recursive* object structures, including the support for class invariants. First we must introduce some basic notion: for arbitrary binary HOL operations *op*, we write $\sigma \vDash P\ op\ Q$ for $\lceil P\,\sigma \rceil op \lceil Q\,\sigma \rceil$. Moreover, we write $\sigma \vDash \partial\ x$ ("x is defined in state σ") for $\mathrm{def}(x\,\sigma)$, and $\sigma \vDash \not\partial\ x$ for the contrary. For constant symbols we will simplify the presentation: for example, we will write 5 for $\lambda\,\sigma.\,5$, $\llcorner \mathrm{true} \lrcorner$ for $\lambda\,\sigma.\,\llcorner \mathrm{true} \lrcorner$, etc.

Now we approach our main goal to provide a type-safe embedding of the accessors, and, consequently, of the whole assertion language.

We define the *store* as a partial map based on the concept of object universes:

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

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

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

Thus we can define type-safe accessor functions, i. e., object identifiers (references) are completely encapsulated. For example, the function for accessing the

left attribute of an object of class Node in a system state σ is defined as follows:

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

For accessor with type set or sequence, we provide definitions that de-reference each element of, e. g., a set of object identifiers and build a set of typed objects.

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

$$\_.\,\text{left} :: V_{(\alpha_C,\beta_C)}((\alpha_C,\beta_C)\,\texttt{Node}) \Rightarrow V_{(\alpha_C,\beta_C)}((\alpha_C,\beta_C)\,\texttt{Node})\,.$$

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 accessor, casts, etc. is also extensible.

All other operations like casting, type- or kind-check are lifted equivalently; in the following, we will always assume these lifted versions since due to our typing discipline, no confusion may arise. These definitions are also generated by our package and "lifted" versions of the theorems are derived.

We turn now to our construction of characteristic sets and the derivation of class invariant theorems. Recall our previous example, where the class `Node` describes a potentially infinite recursive object structure. Assume that we want to constrain the attribute `i` of class `Node` to values greater than 5. This is expressed by the following function approximating the set of possible instances of the class `Node` and its subclasses:

$$\begin{aligned} \text{NodeKindF} :: \mathscr{U}^1_{(\alpha_C,\beta_C,\beta_N)}\,\text{St} \Rightarrow \mathscr{U}^1_{(\alpha_C,\beta_C,\beta_N)}\,\text{St} \Rightarrow (\alpha_C,\beta_C)\,\texttt{Node}\,\text{set} \\ \Rightarrow \mathscr{U}^1_{(\alpha_C,\beta_C,\beta_N)}\,\text{St} \Rightarrow (\alpha_C,\beta_C)\,\texttt{Node}\,\text{set} \\ \text{NodeKindF} \equiv \lambda\,\sigma.\ \lambda\,X.\ \big\{ obj \mid \sigma \vDash \partial\, obj.\text{i} \wedge \sigma \vDash obj.\text{i} > 5 \\ \wedge\, \sigma \vDash \partial\, obj.\text{left} \wedge \sigma \vDash (obj.\text{left}) \in X \\ \wedge\, \sigma \vDash \partial\, obj.\text{right} \wedge \sigma \vDash (obj.\text{right}) \in X \big\} \end{aligned}$$

In a setting with subtyping, we need two characteristic type sets, a more liberal one, the *characteristic kind set*, and narrower one, the *characteristic type set*. By adding the conjunct $\sigma \vDash obj\,\text{isTypeOf(Node)}$ (essentially a notation for the previously defined type tests), we can construct another approximation function (which has obviously the same type as NodeKindF):

$$\begin{aligned} \text{NodeTypeF} \equiv \lambda\,\sigma.\ \lambda\,X.\ \big\{ obj \mid (obj \in (\text{NodeKindF}\ \sigma\ X)) \\ \wedge\, \sigma \vDash obj\,\text{isTypeOf(Node)} \big\} \end{aligned}$$

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

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

For the characteristic type set we proceed analogously. We infer a *class invariant theorem*:

$$\begin{aligned}\sigma \vDash obj \in \text{NodeKindSet} = \sigma \vDash \partial\, obj.\text{i} \wedge \sigma \vDash obj.\text{i} > 5 \\ \wedge\, \sigma \vDash \partial\, obj.\text{left} \wedge \sigma \vDash (obj.\text{left}) \in \text{NodeKindSet} \\ \wedge\, \sigma \vDash \partial\, obj.\text{right} \wedge \sigma \vDash (obj.\text{right}) \in \text{NodeKindSet}\end{aligned}$$

and prove automatically by monotonicity of the approximation functions and their point-wise inclusion:

$$\text{NodeTypeSet} \subseteq \text{NodeKindSet}$$

This kind of theorem remains valid if we add further classes in a class system.

Now we relate class invariants of subtypes to class invariants of supertypes. Here, we use coercion functions described in the previous section; we write $o_{[\text{Node}]}$ for the object o 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:

$$\text{CnodeF} \equiv \lambda\sigma.\ \lambda X.\ \{\, obj \mid obj_{[\text{Node}]} \in (\text{NodeKindF}\ \sigma\ (\lambda o.\, o_{[\text{Node}]}) \setminus X)) \wedge \cdots \}$$

where the ... stand for the constraints specific to the subclass.

Similar to [24] 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 $\mathfrak{C}_N$ ($\mathfrak{K}_N$) and be the characteristic type set (characteristic kind set) of a class `N` and $\mathfrak{C}_C$ and $\mathfrak{K}_N$ 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.:

$$\sigma \vDash obj \in \mathfrak{C}_N \longrightarrow \sigma \vDash obj \in \mathfrak{K}_N$$

and that the kind set of the subclass is (after type coercion) a subset of the type set (and thus also of the kind set) of the superclass:

$$\sigma \vDash obj \in \mathfrak{K}_C \longrightarrow \sigma \vDash obj_{[\text{Node}]} \in \mathfrak{C}_N\,.$$

These proofs are based on 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 were 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.

The Underlying Method. Our object-oriented datatype package also supports a special analysis and verification method based on the idea of providing several versions of invariants that restrict the type and kind sets with different grades. For example, the discussed type sets and kind sets are of major importance when resolving overloading and late-binding: If we can infer from a class invariant that some object must be of a particular *type*, then late-binding method invocation can be reduced to a straight-forward procedure call with simplified semantics.

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

1. a set based on the user-defined invariant,
2. a set allowing undefined references, i. e., all accessor to attributes of type oid are or-ed with a corresponding $\emptyset$-statement, and
3. one allowing undefined references *and* undefined value types, i. e., all accessor to attributes are or-ed with a corresponding $\emptyset$-statement.

This enumeration is ordered ascending with respect to the number of instances that fulfill the conditions, i. e., every object that is in the first set, is also in the other two. Such an hierarchy of invariants allows for formally specifying the circumstances which invariants should hold.

In practice we assume the need for an even more fine-grained graduation of invariants. Whereas at the moment one has to reproduce the encoding process of our package to introduce new invariant types, we intend to provide an automatic mechanism for defining new invariant types, i. e., an interface to our package that defines new type sets and also automatically proves the basic properties, including the inclusion relation with respect to the already defined type sets.

5 A Modular Proof-methodology for Object-oriented Modeling

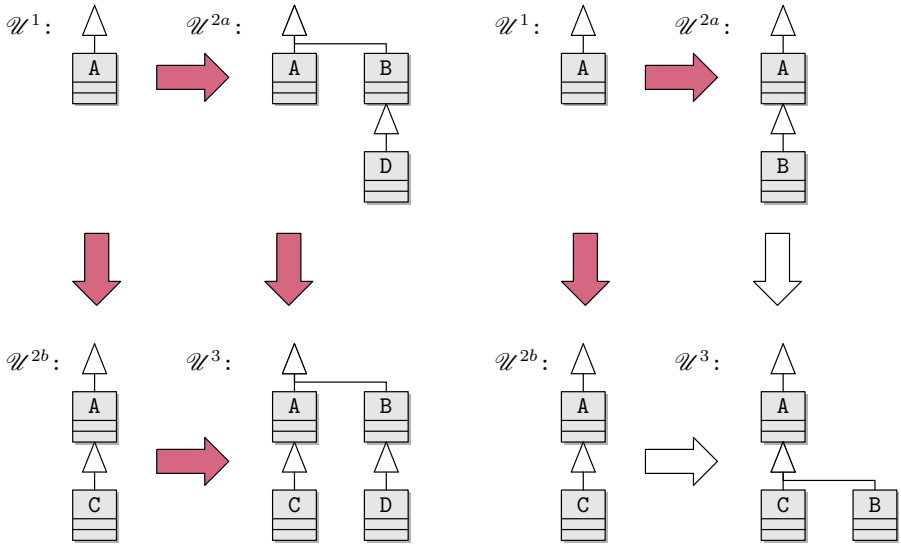
In the previous sections, we discussed a technique to build *extensible* object-oriented data models. Now we turn to the wider goal of a *modular* proof methodology for object-oriented systems and explore achievements and limits of our approach with respect to this goal. Two aspects of modular proofs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

5.1 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



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

Figure 3. Merging Universes

their set of classes including their direct parent classes are disjoint (see Figure 3a). In these cases, there exists a most general type instance of the merged object universe $\mathscr{U}^3$ (the type unifier of both extended universes $\mathscr{U}^{2a}$ and $\mathscr{U}^{2b}$); thus, all theorems built over the extended universes are still valid over the merged universe (see Figure 3a). We argue that the non-overlapping case is the pragmatically more important one; for example, all libraries of the HOL-OCL system [9] are 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.

5.2 Handling Overlapping Merges

Unfortunately, one pragmatically important case in object-oriented modeling is considered as an overlap in our package. Consider the case illustrated in Figure 3b. 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 sub-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 pragmatic 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 proof scripts of either `B` or `C`—usually a matter of a minute. Another option is to introduce an empty class `B'` and 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.

5.3 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 close 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 theorem for this parent class possible).

In usual object-oriented languages, methods can be overridden, method invocations like in object-oriented languages require an overridden resolution mechanism such as *late binding* as used in Java. Late binding uses the dynamic type $\mathrm{isType}_X\ obj$ of *obj*. The late-binding method invocation is notorious for its difficulties for modular proof. Consider the case of an operation:

```
method Node::m()::Bool
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 4 which ensures that a specific dereferentiation is in fact defined, we can specify that a specific dereferentiation $obj.\,\mathrm{left}$ has a given dynamic type. An analogous invariant $\mathrm{Inv}_{\mathrm{left}}(obj)$ can be defined co-inductively. From this invariant, we can directly infer facts like $\mathrm{isType}^{(1)}_{\mathrm{Node}}(obj.\,\mathrm{left})$, and $\mathrm{isType}^{(1)}_{\mathrm{Node}}(obj.\,\mathrm{left}.\,\mathrm{left})$, 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 $\mathrm{Inv}_{\mathrm{left}}(obj)$ 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.

5.4 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 $k :: \text{nat}$ that computes this object set only up to the length k of the paths in the path set.

$$\begin{aligned}
&\text{ObjSetA}_{\text{left}}\ 0\ obj\ \sigma = \{\}\\
&\text{ObjSetA}_{\text{left}}\ k\ obj\ \sigma = \text{if}\ \sigma \models \partial\ obj\ \text{then}\{\}\\
&\qquad\qquad\qquad\qquad \text{else}\ \{obj\} \cup \text{ObjSetA}_{\text{left}}\ (k-1)\ (obj.\,\text{left}\ \sigma)\ \sigma
\end{aligned}$$

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

$$\bigcup_{n \in \text{Nat}} \text{ObjSetA}_{\text{left}}\ n\ obj\ \sigma\,.$$

On the other hand, we can add a *state invariant* on our concept of state per type definition $\alpha\,\text{St} = \{\sigma :: \text{oid} \rightharpoonup \mathscr{U}^{\alpha}.\,\text{Inv}\ \sigma\}$. Here, we require for *inv* that each oid points to an object that contains itself:

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

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.

6 Application: A Shallow Embedding of IMP++

In the following, we integrate the operations derived from an object-oriented data model into assertions in a derived Hoare-Calculus for a small, imperative language. This language is pretty much in the spirit of Featherweight Java [14], 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 language; demonstrating how this definitions scale up with the presented machinery to a modular proof method, however, 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 not further complicating its rules by side-conditions related to well-formedness of objects, the syntactic admissibility of attribute accesses to an object or reasoning along

the class hierarchy as in the deep embedding of, e. g., NanoJava [23]. We will show that compact calculi for denotational, operational and axiomatic semantics can be derived in a standard exercise.

We follow deliberately the standard presentation of IMP [20], a canonical imperative core language, in the Isabelle/HOL library; this language has been inspired by a standard textbook on program semantics [26]. We will extend IMP with object-oriented typedness, creation, update, selection and a simple form for exceptional computation (motivated by illegal memory accesses). In a small example, we sketch how to apply it for reasoning on weak and strong data invariants on tree-like structures.

In contrast to the previous sections where definitions and proofs were done automatically for all classes and attributes—the proof presented here are done interactively. However, we emphasize that a large part of it (e. g., the core Hoare-Calculus and the rules for update and create) could be mechanically derived, too.

6.1 Syntax

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

$$\begin{aligned}\alpha\ \mathrm{com} = \ & \texttt{SKIP} && | \ \texttt{EXN} \\ | \ & \texttt{CMD}\ \alpha\ \mathrm{cmd} && | \ \texttt{IF}\ \alpha\ \mathrm{bexp}\ \texttt{THEN}\ \alpha\ \mathrm{com}\ \texttt{ELSE}\ \alpha\ \mathrm{com} \\ | \ & \alpha\ \mathrm{com}\ ;\ \alpha\ \mathrm{com} && | \ \texttt{WHILE}\ \alpha\ \mathrm{bexp}\ \texttt{DO}\ \alpha\ \mathrm{com}\end{aligned}$$

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 α state $\Rightarrow$ α 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 α state $\Rightarrow$ bool$_\perp$. Any assertion has a type which is an instance of α bexp, thus, it can be used as control expression in IMP++.

6.2 Denotational Semantics

In general, 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 $(\alpha :: \mathrm{bot}\ \mathrm{state}_\perp \times \alpha\ \mathrm{state}_\perp)\mathit{set}$. 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:

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

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

$$\begin{aligned}
C(\texttt{SKIP}) &= \mathrm{Id} \\
C(\texttt{EXN}) &= \{(s,t).\, t = \bot\} \\
C(\texttt{CMD}\ f) &= \{(s,t).\, s = \bot \wedge t = \bot\} \cup \{(s,t).\, \mathrm{def}\, s \wedge t = f\, \lceil s \rceil\} \\
C(c_0; c_1) &= C(c_1) \circ_\bot C(c_0) \\
C(\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2) &= \{(s,t).\, (s = \bot \vee b\, \lceil s \rceil = \bot) \wedge t = \bot\} \\
&\quad \cup \{(s,t).\, \mathrm{def}\, s \wedge b\, \lceil s \rceil = \lfloor \mathrm{true} \rfloor \wedge (s,t) \in C\, c_1\} \\
&\quad \cup \{(s,t).\, \mathrm{def}\, s \wedge b\, \lceil s \rceil = \lfloor \mathrm{false} \rfloor \wedge (s,t) \in C\, c_2\} \\
C(\texttt{WHILE}\ b\ \texttt{DO}\ c) &= \mathrm{lfp}(\Gamma\, b\, (C\, c))
\end{aligned}$$

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

$$\begin{aligned}
\Gamma\, b\, cd \equiv (\lambda\, \phi.\, &\{(s,t).\, (s = \bot \vee b\, \lceil s \rceil = \bot) \wedge t = \bot\} \cup \\
&\{(s,t).\, \mathrm{def}\, s \wedge b\, \lceil s \rceil = \lfloor \mathrm{true} \rfloor \wedge (s,t) \in (\phi \circ_\bot cd)\} \cup \\
&\{(s,t).\, \mathrm{def}\, s \wedge b\, \lceil s \rceil = \lfloor \mathrm{false} \rfloor \wedge s = t\})
\end{aligned}$$

6.3 Hoare Semantics

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:

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

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 $\models$ 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 $\models \{\odot P'\}c\{\odot Q'\}$ relate "non-exception" states allowing inference of normal behavior. The derived calculus is then surprisingly standard (see Table 1).

6.4 Data-model Specific Hoare Rules

Recall our running example with the classes `Node` and `CNode`. Besides the type-safe accessor functions, we need families of store-related (i. e., level 1) update and creation operations on objects. For example, the lifting of update operations to the level of the assertion language is straight-forward:

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

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

$$\frac{\forall s.\, P'\, s \longrightarrow P\, s \quad \vDash \{P\}c\{Q\} \quad \forall s.\, Q\, s \longrightarrow Q'\, s}{\vDash \{P'\}c\{Q'\}} \qquad \frac{}{\vDash \{\odot P\}\,\texttt{SKIP}\{\odot P\}}$$

$$\frac{\vDash \{\odot P\}c\{\odot Q\} \quad \vDash \{\odot Q\}d\{\odot R\}}{\vDash \{\odot P\}c; d\{\odot R\}} \qquad \frac{\vDash \{\odot\, \lambda\, \sigma.\, P\sigma \wedge (\ulcorner\sigma\urcorner \vDash b)\}c\{\odot P\}}{\vDash \{\odot P\}\{\texttt{WHILE}\}b\{\texttt{DO}\}c\{\odot\, \lambda\, \sigma.\, P\sigma \wedge (\ulcorner\sigma\urcorner \vDash \neg\, b)\}}$$

$$\frac{}{\vDash \{\lambda\, \sigma.\, \sigma = \bot\}\, c\, \{\lambda\, \sigma.\, \sigma = \bot\}} \qquad \frac{}{\vDash \{\odot\, \lambda\, \sigma.\, \ulcorner\sigma\urcorner \vDash \partial\, f \wedge Q(f\, \ulcorner\sigma\urcorner)\}\, \texttt{CMD}\, f\{\odot\, Q\}}$$

$$\frac{\begin{array}{c}\vDash \{\odot\, \lambda\, \sigma.\, (P\sigma) \wedge (\ulcorner\sigma\urcorner \vDash b) \wedge (\ulcorner\sigma\urcorner \vDash \partial\, b)\}d\{\odot\, Q\} \\ \vDash \{\odot\, \lambda\, \sigma.\, (P\sigma) \wedge (\ulcorner\sigma\urcorner \vDash \neg\, b) \wedge (\ulcorner\sigma\urcorner \vDash \partial\, b)\}d\{\odot\, Q\}\end{array}}{\vDash \{\odot\, P\}\{\texttt{IF}\}b\{\texttt{THEN}\}c\{\texttt{ELSE}\}d\{\odot\, Q\}}$$

Table 1. The Hoare Calculus for IMP++

With respect to the creation operations, we define:

$$\text{newOid}\, \sigma \equiv \varepsilon\, x.\, x \notin \text{dom}\, \sigma$$

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

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

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

$$\begin{aligned}\mathit{obj}\,.\,\text{new}^{(1)}_{\text{left}}\ \equiv \lambda\, \sigma.\, \text{let}\, \sigma' = \sigma(\text{newOid}\, \sigma := \text{new}_{\text{Node}}\ (\text{newOid}\, \sigma)) \\ \text{in}\ \mathit{obj}.\,\text{set}^{(1)}_{\text{left}}\ (\text{new}_{\text{Node}}\, (\text{newOid}\, \sigma))\sigma'\end{aligned}$$

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

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

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

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

6.5 An Example in IMP++.

A program that produces the smallest possible object system satisfying the CNode invariant looks in a fictive 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 call 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}.

The stack-object class class m_{so} has the form:

```
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).

We want to specify that the program establishes by a sequence of creation and update steps the global invariant verification of the body is stated as follows, assuming that the stack object is defined when the method is called:

$$\models \{\odot\, \lambda\, \sigma.\, \sigma \models \partial(m_{\text{so}})\} m_{\text{body}} \{\odot\, \sigma \models m_{\text{so}}.\, \text{return}^{(1)} \in \text{CNodeKindSet}\}$$

The proof for this statement proceeds in essentially two phases: First, by several applications of the consequence rule and the update-rule, we accumulate an

equation system as assertion:

$$\begin{aligned}
& \sigma \models \partial(m_{\mathrm{so}}.\,\mathrm{H1}^{(1)}) \\
\wedge\ & \sigma \models \partial(m_{\mathrm{so}}.\,\mathrm{H2}^{(1)}) \\
\wedge\ & \sigma \models m_{\mathrm{so}}.\,\mathrm{H1}.\,\mathrm{i}^{(1)} = 7 \wedge \sigma \models m_{\mathrm{so}}.\,\mathrm{H1}.\,\mathrm{color}^{(1)} = \llcorner\mathrm{true}\lrcorner \\
\wedge\ & \sigma \models m_{\mathrm{so}}.\,\mathrm{H1}.\,\mathrm{left}^{(1)} = H2 \wedge \sigma \models m_{\mathrm{so}}.\,\mathrm{H1}.\,\mathrm{right}^{(1)} = H2 \\
\wedge\ & \sigma \models m_{\mathrm{so}}.\,\mathrm{H2}.\,\mathrm{i}^{(1)} = 9 \wedge \sigma \models m_{\mathrm{so}}.\,\mathrm{H2}.\,\mathrm{color}^{(1)} = \llcorner\mathrm{false}\lrcorner \\
\wedge\ & \sigma \models m_{\mathrm{so}}.\,\mathrm{H2}.\,\mathrm{left}^{(1)} = H1 \wedge \sigma \models m_{\mathrm{so}}.\,\mathrm{H2}.\,\mathrm{right}^{(1)} = H1 \\
\wedge\ & \sigma \models m_{\mathrm{so}}.\,\mathrm{return}^{(1)} = H1
\end{aligned}$$

(Recall that we overload $7, 9, \ldots$ with $\lambda\,\sigma.7$, $\lambda\,\sigma.7, \ldots$ to simplify notation). This assertion must imply the postcondition, which is reduced to:

$$\sigma \models m_{\mathrm{so}}.\,\mathrm{return}^{(1)} \in \mathrm{gfp}\ \mathrm{CnodeKindF}$$

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

$$\frac{\begin{array}{c}[\sigma \models m_{\mathrm{so}}.\,\mathrm{return}^{(1)} \in X] \\ \vdots \\ \bigwedge X.\quad \sigma \models m_{\mathrm{so}}.\,\mathrm{return}^{(1)} \in \mathrm{CnodeKindF}\ X\end{array}}{\sigma \models m_{\mathrm{so}}.\,\mathrm{return}^{(1)} \in \mathrm{gfp}\ \mathrm{CnodeKindF}}$$

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. In [15] 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.

7 Conclusion

We presented an extensible universe construction supporting object-oriented datatype theories including subtyping and (single) inheritance. On the theoretical side, this proves that object-oriented datatype theories can be represented in typed λ-calculus with Hindley-Milner Polymorphism. As a by-product, the construction also gives insight into the representation of open-world and closed-world assumptions in types. The achievement on the practical side is three-fold: First, we show that the core of object-oriented reasoning can be made amenable to off-the-shelf HOL theorem provers (no Isabelle specific features are inherently necessary for this) in form of a shallow embedding. Second, this can be done in a conservative way: provided that the 9 axioms of HOL are consistent (on which the large majority of logicians agree), the generated datatype theory will also be consistent. Third, albeit the underlying complexity, deriving automatically

the datatype theory from the basic definitions is still technically feasible: [7,9] report on an example suite of class models. The computation time for each of these models is below 2 minutes on recent hardware.

One might object that the universe construction described in Section 3 and Section 4 is entirely meta-theoretic, thus not verifiable; and principles like conservative definitions are not applicable on this level. However, this is not entirely true. While concepts like "the set of all HOL-types" or "the set of class types" are indeed not formalized in HOL, for each concrete type resulting from the construction a consistent theory is generated. If our construction or our implementation has an error, Isabelle will refuse to accept these definitions or the proofs.

Related Work. Work on object-oriented semantics based on deep embeddings has been discussed earlier. For shallow embeddings, to the best of our knowledge, there is only [25]. In this approach, however, emphasis is put on a universal type for a *classes* comprising method tables. This results in local "universes" for input and output types of methods and the need for reasoning on class isomorphisms. Subtyping on *objects* must be expressed implicitly via refinement. 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 [19]. Since only α-extensions are used, this results in a restricted form of class types with no coercion mechanism to α `Object`.

Datatype packages have been considered mostly in the context of HOL or functional programming languages. Going back to ideas of Milner in the 70ies, systems like [17,5] build over an S-expression like term universe (co)-inductive sets which are abstracted to (freely generated) datatypes. Paulson's inductive package [24] also uses subsets of the ZF set universe i.

Even systems like Spec# [3,15] or Krakatoa [16], which are clearly more advanced with respect to the degree of automation for *program verification* as a whole, might profit from guaranteed consistent data-models: at present, a quite substantial axiomatization of a given object-oriented memory model is generated in these systems. The second author witnessed several logical inconsistencies in the data model underlying Spec#. We believe that the properties of our object-oriented data model, even if taken axiomatically, could provide assurance. If required, our system can generate for given class system proofs of consistency.

Future Work. We see the following lines of future research:

- *Multiple Inheritance.* Our approach is strictly limited to single inheritance. However, it is easy to extend our package with support for multiple subtyping based on *interfaces*.
- *Modular Verification of Recursive Methods.* The presented language does not comprise method invocation—it remains to be shown how the presented machinery can be used for an extensible program theory comprising these crucial features.
- *Support for Inductive Constraints.* By introducing measure-functions over object-structures, inductive datatypes can be characterized for defined mea-

sures 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.

References

1. Unified modeling language specification (version 1.5) (2003). Available as OMG document formal/03-03-01
2. Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof, 2nd edn. (2002)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: G. Barthe, L. Burdy, M. Huisman, J.L. Lanet, T. Muntean (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), vol. 3362, pp. 49–69 (2005). DOI 10.1007/b105030
4. Basin, D.A., Kuruma, H., Takaragi, K., Wolff, B.: Verification of a signature architecture with HOL-Z. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) FM 2005: Formal Methods, vol. 3582, pp. 269–285 (2005). DOI 10.1007/11526841_19
5. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—lessons learned in formal-logic engineering. In: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (eds.) Theorem Proving in Higher Order Logics (TPHOLs), vol. 1690, pp. 19–36 (1999). DOI 10.1007/3-540-48256-3_3
6. Bierman, G.M., Parkinson, M.J.: Effects and effect inference for a core Java calculus **82**(7), 1–26 (2003). DOI 10.1016/S1571-0661(04)80803-X
7. Brucker, A.D.: An interactive proof environment for object-oriented specifications. Ph.D. thesis, ETH Zurich (2007). URL `http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007`. ETH Dissertation No. 17097.
8. Brucker, A.D., Rittinger, F., Wolff, B.: HOL-Z 2.0: A proof environment for Z-specifications. Journal of Universal Computer Science **9**(2), 152–172 (2003). URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-z-2003`
9. Brucker, A.D., Wolff, B.: The HOL-OCL book. Tech. Rep. 525, ETH Zurich (2006). URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006`
10. Brucker, A.D., Wolff, B.: HOL-OCL – A Formal Proof Environment for UML/OCL. In: J. Fiadeiro, P. Inverardi (eds.) Fundamental Approaches to Software Engineering (FASE08), 4961, pp. 97–100 (2008). DOI 10.1007/978-3-540-78743-3_8. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-2008`
11. Drossopoulou, S., Eisenbach, S.: Describing the semantics of Java and proving type soundness. In: J. Alves-Foss (ed.) Formal Syntax and Semantics of Java, vol. 1523, pp. 41–82 (1999). DOI 10.1007/3-540-48737-9_2
12. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. In: J. Alves-Foss (ed.) Formal Syntax and Semantics of Java, vol. 1523, pp. 241–269 (1999). DOI 10.1007/3-540-48737-9_7
13. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: a theorem proving environment for higher order logic (1993)

14. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ **23**(3), 396–450 (2001). DOI 10.1145/503502.503505
15. Leino, K.R.M., Müller, P.: Modular verification of static class invariants. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) FM 2005: Formal Methods, vol. 3582, pp. 26–42 (2005). DOI 10.1007/11526841_4
16. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In: J. Hurd, T.F. Melham (eds.) Theorem Proving in Higher Order Logics (TPHOLs), vol. 3603, pp. 179–194 (2005). DOI 10.1007/11541868_12
17. Melham, T.F.: A package for inductive relation definitions in HOL. In: M. Archer, J.J. Joyce, K.N. Levitt, P.J. Windley (eds.) International Workshop on the HOL Theorem Proving System and its Applications (TPHOLs), pp. 350–357 (1992)
18. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF **9**(2), 191–223 (1999). DOI 10.1017/S095679689900341X
19. Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in higher-order logic. In: J. Grundy, M.C. Newey (eds.) Theorem Proving in Higher Order Logics (TPHOLs), vol. 1479, pp. 349–366 (1998). DOI 10.1007/BFb0055146
20. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook **10**(2), 171–186 (1998). DOI 10.1007/s001650050009
21. Nipkow, T., von Oheimb, D.: Java$_{light}$ is type-safe—definitely. In: ACM Symp. Principles of Programming Languages (POPL), pp. 161–170 (1998). DOI 10.1145/268946.268960
22. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, vol. 2283 (2002). DOI 10.1007/3-540-45949-9
23. von Oheimb, D., Nipkow, T.: Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In: L.H. Eriksson, P.A. Lindsay (eds.) FME 2002: Formal Methods—Getting IT Right, vol. 2391, pp. 89–105 (2002). DOI 10.1007/3-540-45614-7_6
24. Paulson, L.C.: A fixedpoint approach to (co)inductive and (co)datatype definitions. In: G. Plotkin, C. Stirling, M. Tofte (eds.) Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 187–211 (2000)
25. Smith, G., Kammüller, F., Santen, T.: Encoding Object-Z in Isabelle/HOL. In: D. Bert, J.P. Bowen, M.C. Henson, K. Robinson (eds.) ZB 2002: Formal Specification and Development in Z and B, vol. 2272, pp. 82–99 (2002). DOI 10.1007/3-540-45648-1_5
26. Winskel, G.: The Formal Semantics of Programming Languages (1993)