Extensible Object-oriented Data Models in Isabelle/HOL

Achim D. Brucker and Burkhart Wolff

Information Security, ETH Zurich, 8092 Zurich, Switzerland
{brucker, bwolff}@inf.ethz.ch

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 enables 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 system, to a theory containing object-universes, constructors, and accessor functions, coercions (casts) between dynamic and static types, characteristic sets, their relations reflecting inheritance, and co-inductive class invariants. The package is conservative, i. e., all properties are derived entirely from constant definitions. As an application, we show constraints over object structures.

1 Introduction

While object-oriented (OO) 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 programs on the one hand and object-oriented programs on the other. This is mirrored in the increasing limitations of proof environments.

The existing proof environments dealing with subtyping and references can be categorized as: 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# [2,12], Krakatoa [13] and several similar tools based on the Java Modeling Language (JML). The underlying idea is to compile object-oriented programs into standard imperative ones and to apply a verification condition generator on the latter. While technically sometimes very advanced, the foundation of these tools is quite problematic: The generators usually supporting a large language fragment are not verified, and it is not clear if the generated conditions are sound and complete 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., [9,10,19,23]). 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 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 [16] or for HOL-Z [6,3].

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

At first sight, it seems impossible to apply the injective shallow embedding technique to object-oriented languages: Their characteristic features like subtyping and inheritance are not present in the typed λ-calculi underlying HOL systems. However, an injective mapping does 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, this requires a data model that respects semantic properties like no loss of information in casts.

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 to an existing class system a new class without reproving everything. The problem becomes apparent when considering the underlying *state* of an object-oriented program 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. Object structures (or: states) are maps of type $\mathrm{oid} \Rightarrow \mathscr{U}$ 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 system. 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 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 systems, 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 kind of logic compiler that generates for each class system and its extensions conservative definitions. This includes the definition of constructors and accessors, casts between types, tests, characteristic sets of objects. On this basis, properties reflecting subtyping and proof principles like class invariants are automatically derived.

2 Formal and Technical Background

Isabelle [18] 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 offers support for checks for conservatism of definitions, datatypes, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableau provers.

Higher-order logic (HOL) [1] 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 $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ 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.

We assume a type class $\alpha ::$ bot for all types α that provide an exceptional element $\bot$; for each type in this class a test for defindness 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 $\llcorner\_\lrcorner : \alpha \Rightarrow \alpha_\bot$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\bot \Rightarrow \alpha$ its inverse for defined values. Partial functions $\alpha \rightharpoonup \beta$ are just functions $\alpha \Rightarrow \beta$option.

3 Level 0: Typed Object Universes

In this section, we introduce our 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* 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 "living" in $\mathscr{U}^i$ remain valid in $\mathscr{U}^{i+1}$.

3.1 A Formal Framework of Object Structure Encoding

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 a derived class `Cnode` (thus, `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. This set, although denoted in usual set-notation, is a meta-theoretic construct, i. e., it cannot be formalized in HOL .

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

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

Attributes with class types, e. g., the attribute `left` of class `Node`, are encoded using the abstract type oid. These object identifiers (i. e., references) will be resolved by accessor functions like $A\,.\,\text{left}^{(1)}$ for a given state; an access failure will be reported by $\bot$.

In principle, a class is a 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 are one of the reasons why we can built 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. 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 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. For example, for OCL (Object Constraint Language), this is `OclAny`, for Java this is `Object`.

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* α `Object` $:= \big((\text{Object}_t \times \text{oid}) \times \alpha_\bot\big)$.

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 7.4). 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* $\mathfrak{U}_{ref}$ *resp.* $\mathfrak{U}_{nref}$ *(abbreviated* $\mathfrak{U}_x$*) is inductively defined by:*

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

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 $\mathscr{U}_\alpha^0 = \alpha$ `Object`. However, we will need the ability to store *Values* = `Real`+`Integer`+`Boolean`+`String`. Therefore, we define the initial universe type by $\mathscr{U}_\alpha^0 = \alpha$ `Object` + *Values*. Extending the initial universe $\mathscr{U}_{(\alpha)}^0$, in parallel, with the classes `Node` and `Cnode` leads to the following universe type:

$$\mathscr{U}^1_{(\alpha_C,\beta_C,\beta_N)} = \Big((\mathrm{Node_t} \times \texttt{Integer} \times \mathrm{oid} \times \mathrm{oid}) \times \big((\mathrm{Cnode_t} \times \texttt{Boolean}) \times (\alpha_C)_\perp + \beta_C\big)_\perp + \beta_N\Big)\ \texttt{Object} + \mathit{Values}\,.$$

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

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

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

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

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 constructor may be undefined.

3.2 Handling Instances

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

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

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

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

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`:

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

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

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

For attributes with object types we return an *oid*. In Section 5, we show how these projectors can be used for defining a type-safe variant.

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

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

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 $\bot$:

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

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

$$\mathrm{isType}^{(0)}_{\mathrm{Object}}\ \mathit{self} = \mathrm{isKind}^{(0)}_{\mathrm{Object}} \wedge \neg\big((\mathrm{def} \circ \mathrm{base})\ \mathit{self}\big) \quad \text{of type } \alpha\ \texttt{Object} \Rightarrow \mathrm{bool}.$$

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

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

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

These type-casts are changing the *static type* of an object, while the *dynamic type* remains unchanged.

Note, for a universe construction without values, e. g., $\mathscr{U}^0_\alpha = \alpha\ \texttt{Object}$, the universe type and the class type for the common supertype are the same. In that case there is a particularly strong relation between class types and universe types on the one hand and on the other there is a strong relation between the conversion functions and the injections and projections function. In more detail, one can understand the projections as a cast from the universe type to the given class type and the injections 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 constructions establishes a subtype relation via inheritance. Therefore, a set of Nodes (with type $\big((\alpha_1, \beta)\,\mathrm{Node}\big)\ \texttt{Set}$) can also contain objects of type `Cnode`. For resolving operation overloading, i. e., late-binding, the packages generates operation tables user-defined operations; see [7,5] for details.

4 Properties of the Object Encoding

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 [20]). These theorems are proven for each model, e. g., during loading a

specific class system. 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 a model with the classes `A` and `B` where `B` is a subclass of `A`. We start by proving this subtype relation for both our class type and universe type representation:

$$\frac{\mathrm{isUniv}_{\mathrm{A}}^{(0)}\, u}{\mathrm{isUniv}_{\mathrm{B}}^{(0)}\, u} \quad (1.\mathrm{a}) \qquad \frac{\mathrm{isType}_{\mathrm{B}}^{(0)}\, \mathit{self}}{\mathrm{isKind}_{\mathrm{A}}^{(0)}\, \mathit{self}} \quad (1.\mathrm{b})$$

We also show that our conversion between universe representations and object representation is satisfy the *no loss of information in casts*-principle and that both type systems are isomorphic:

$$\frac{\mathrm{isUniv}_{\mathrm{A}}^{(0)}\, u}{\mathrm{mk}_{\mathrm{A}}^{(0)}(\mathrm{get}_{\mathrm{A}}^{(0)}\, u) = u} \quad (2.\mathrm{a}) \qquad \frac{\mathrm{isType}_{\mathrm{A}}^{(0)}\, \mathit{self}}{\mathrm{get}_{\mathrm{A}}^{(0)}(\mathrm{mk}_{\mathrm{A}}^{(0)}\, \mathit{self}) = \mathit{self}} \quad (2.\mathrm{b})$$

$$\frac{\mathrm{isType}_{\mathrm{B}}^{(0)}\, \mathit{self}}{\mathrm{isUniv}_{\mathrm{A}}^{(0)}(\mathrm{mk}_{\mathrm{A}}^{(0)}\, \mathit{self})} \quad (3.\mathrm{a}) \qquad \frac{\mathrm{isUniv}_{\mathrm{A}}^{(0)}\, u}{\mathrm{isType}_{\mathrm{A}}^{(0)}(\mathrm{get}_{\mathrm{A}}^{(0)}\, u)} \quad (3.\mathrm{b})$$

Moreover, that we can "re-cast" an objects safely, i. e., the dynamic (class) type of an object can be casted to a supertype and back:

$$\frac{\mathrm{isType}_{\mathrm{B}}^{(0)}\, \mathit{self}}{\mathrm{isType}_{\mathrm{B}}^{(0)}\left(\left(\left(\mathit{self}_{[\mathrm{A}]}^{(0)}\right)_{[\mathrm{B}]}^{(0)}\right)_{[\mathrm{A}]}^{(0)}\right)} \quad (4)$$

The datatype package also shows similar properties for the injections and projections into attributes.

5 Level 1: A Type-safe Object Store

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

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

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:

$$\mathit{self}\,.\,\mathrm{left}^{(1)}\ \sigma \equiv \begin{cases} \mathrm{get}_{\mathrm{Node}}^{(0)}\, u & \text{if } \sigma((\mathit{self}\ \sigma)\,.\,\mathrm{left}^{(0)}) = \mathrm{Some}\, u \\ \bot & \text{else} \end{cases}$$

The object language type `.left : Node -> Node` is now represented by our construction with type $V_{(\alpha_C,\beta_C)}\big((\alpha_C,\beta_C)\ \texttt{Node}\big) \Rightarrow V_{(\alpha_C,\beta_C)}\big((\alpha_C,\beta_C)\ \texttt{Node}\big)$. 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 analogously; here we present only the case of the cast:

$$self^{(1)}_{[\mathrm{A}]}\ \sigma \equiv (self\ \sigma)^{(0)}_{[\mathrm{A}]}$$

Moreover, the properties of the previous section were reformulated for this level.

With accessors and cast operations, we have now a path-language to access specific values in object structures. On top of this path language, we add a small assertion language to express properties: We write $\sigma \vDash \partial\ x$ for $\mathrm{def}(x\ \sigma)$, and $\sigma \vDash \not\partial\ x$ for the contrary. With this predicate we can specify that the access to a value along a path succeeds in a given state.

Moreover, for arbitrary binary HOL operations *op* such as $\_ = \_$, $\_ <= \_$, $\_ \subseteq \_$,$\_ \wedge \_$, $\_ \rightarrow \_$, ..., we write $\sigma \vDash P\ op\ Q$ for $\ulcorner P\ \sigma \urcorner\ op\ \ulcorner Q\ \sigma \urcorner$. Note that $\ulcorner \_ \urcorner$ is underspecified for $\bot$, thus for illegal access into the state. An alternative semantic choice for the assertion language consists in a three-valued logic (cf.[7]).

6 Level 2: Co-inductive Properties in Object Structures

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

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:

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

In a setting with subtyping, we need two characteristic type sets, a sloppy one, the *characteristic kind set*, and a fussy one, the *characteristic type set*. By adding the conjunct $\sigma \vDash \mathrm{isType}^{(1)}_{\mathrm{Node}}\ self$, we can construct another approximation function (which has obviously the same type as NodeKindF):

$$\begin{aligned}\mathrm{NodeTypeF} \equiv \lambda\,\sigma.\ \lambda\,X.\ \big\{ self \mid (self \in (\mathrm{NodeKindF}\ \sigma\ X))\\ \wedge\ \sigma \vDash \mathrm{isType}^{(1)}_{\mathrm{Node}}\ self\big\}\end{aligned}$$

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*:

$$\begin{aligned}\sigma \vDash \mathit{self} \in \text{NodeKindSet} = \sigma \vDash \partial\,\mathit{self}\,.\,\text{i}^{(1)} \wedge \sigma \vDash \mathit{self}\,.\,\text{i}^{(1)} > 5\\ \wedge\ \sigma \vDash \partial\,\mathit{self}\,.\,\text{left}^{(1)} \wedge \sigma \vDash (\mathit{self}\,.\,\text{left}^{(1)}) \in \text{NodeKindSet}\\ \wedge\ \sigma \vDash \partial\,\mathit{self}\,.\,\text{right}^{(1)} \wedge \sigma \vDash (\mathit{self}\,.\,\text{right}^{(1)}) \in \text{NodeKindSet}\end{aligned}$$

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

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

This kind of theorems 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 cast functions described in the previous section; we write $\mathit{self}^{(1)}_{[\text{Node}]}$ 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:

$$\begin{aligned}&\text{CnodeF} \equiv \lambda\,\sigma.\ \lambda\,X.\\ &\qquad \left\{\mathit{self} \;\middle|\; \mathit{self}^{(1)}_{[\text{Node}]} \in (\text{NodeKindF}\ \sigma\ (\lambda\,\mathit{obj}.\ \text{obj}^{(1)}_{[\text{Node}]})\ \text{`}\ X)) \wedge \cdots\right\}\end{aligned}$$

where the ... stand for the constraints specific to the subclass and ` denotes the pointwise application.

Similar to [4] or [21] 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.:

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

And also, 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:

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

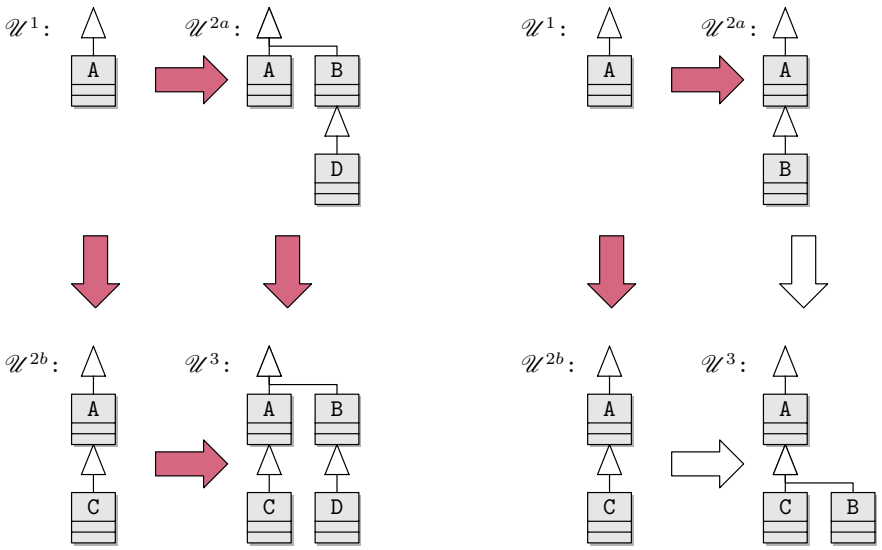
These proofs are based on co-inductions and involve a kind of bi-simulation of (potentially) infinite object structures. Further, these proofs depend on theorems that are already proven over the pre-defined types, e. g., `Object`. These proofs where done in the context of the initial universe $\mathscr{U}^0$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

7 A Modular Proof-Methodology for OO Modeling

In the previous sections, we discussed a technique to build *extensible* object-oriented data models. Now we turn to the wider goal of an *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 systems 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.



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

Figure 1. Merging Universes

7.1 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 including their direct parent classes are disjoint (see Figure 1a). 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 1a). We argue that the non-overlapping case is the pragmatically more important one; for example, all libraries of the HOL-OCL system [7] 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.

7.2 Handling Overlapping Merges

Unfortunately, there is a pragmatically important case in object-oriented modeling that will by considered as an overlap in our package. Consider the case illustrated in Figure 1b. 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 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.

7.3 Modularity in an Open-world: Dynamic Typing

Our notion of extensible class systems 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 lemma for this parent class possible).

Since methods can be overloaded, method invocations like in object-oriented languages require an overloading resolution mechanism such as *late binding* as

used in Java. Late binding uses the dynamic type $\mathrm{isType}^{(1)}_{\mathrm{X}}\ \mathit{self}$ of *self*. The late-binding method invocation is notorious for its difficulties for modular proof. Consider the case of an operation:

```
method Node::m()::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 6 which ensures that a specific de-referentiation is in fact defined, it is possible to specify that a specific de-referentiation $\mathit{self}\,.\,\mathrm{left}^{(1)}$ has a given dynamic type. An analogous invariant $\mathrm{Inv}_{\mathrm{left}}(\mathit{self})$ can be defined co-inductively. From this invariant, we can directly infer facts like $\mathrm{isType}^{(1)}_{\mathrm{Node}}\,(\mathit{self}\,.\,\mathrm{left}^{(1)})$, and $\mathrm{isType}^{(1)}_{\mathrm{Node}}\,(\mathit{self}\,.\,\mathrm{left}^{(1)}\,.\,\mathrm{left}^{(1)})$, i. e., in an object graph satisfying this invariant, the left "spine" must consist of nodes of dynamic type `Node`. Strengthening the precondition `P` by $\mathrm{Inv}_{\mathrm{left}}(\mathit{self})$ consequently allows to establish postcondition `Q`—in a modular manner, since only the method implementation above has to be considered in the proof. Invoking the method on an object graph that does not satisfy this invariant can therefore be considered as a breach of the contract.

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

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

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

$$\bigcup_{n \in \mathrm{Nat}} \mathrm{ObjSetA}_{\mathrm{left}}\ n\ \mathit{self}\ \sigma\,.$$

On the other hand, we can add an *state invariant* on our concept of state per type definition $\alpha\,\mathrm{St} = \{\sigma :: \mathrm{oid} \rightharpoonup \mathscr{U}^{\alpha}.\,\mathrm{Inv}\ \sigma\}$. 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.

8 Conclusion

We presented an extensible universe construction supporting object-oriented features such as subtyping and (single) inheritance. 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, but not discussed due to space limitations.

The universe-construction is supported by a package (developed as part of the HOL-OCL project [7]). 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 1 describes the size of each of the above

	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 1. Importing Different UML/OCL Specifications.

mentioned models together with the number of generated theorems and the time needed for importing them into HOL-OCL. The number of generated theorems depends linearly on the number of classes, attributes, associations, operations and OCL constraints. For generalizations, a quadratic number (with respect to the number of classes in the model) of casting definitions have to be generated and also a quadratic number of theorems have to be proven. The time for encoding the models depends on the number of theorems generated and also on the complexity on their complexity.

8.1 Related Work

Datatype packages have been considered mostly in the context of HOL or functional programming languages. Systems like [14,4] build over a S-expression like term universe (co)-inductive sets which are abstracted to (freely generated) datatypes. Paulsons inductive package [21] also uses subsets of the ZF set universe i.

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 [22]. 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. 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 [17]. Since only α-extensions are used, this results in a restricted form of class types with no cast mechanism to α `Object`.

The underlying encoding used by the loop tool [11] and Jive [15] shares same basic ideas with respect to the object model. However, the overall construction based on a closed-world assumption and thus, not extensible. The support for class invariants is either fully by hand or axiomatic.

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 the semantic infrastructure of of other target languages is required to make our package more widely applicable.
- *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 in the Isabelle library, Hoare-calculi for object-oriented-programs can be derived (as presented in [8]). On this basis, verification condition generators can be proven sound and to a certain extent, complete. This leads to effective program verification techniques based entirely on derived rules.

References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, *LNCS*, vol. 3362, pp. 49–69. Springer, 2004.
3. D. Basin, H. Kuruma, K. Takaragi, and B. Wolff. Verification of a signature architecture with HOL-Z. In *FM*, *LNCS*, vol. 3582, pp. 269–285. Springer, 2005.
4. S. Berghofer and M. Wenzel. Inductive datatypes in HOL—lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, eds., *TPHOLs*, *LNCS*, vol. 1690, pp. 19–36. Springer, 1999.

5. A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. Ph.D. thesis, ETH Zurich, 2007. ETH Dissertation No. 17097.
6. A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.
7. A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zürich, 2006.
8. A. D. Brucker and B. Wolff. A package for extensible object-oriented data models with an application to IMP++. In A. Roychoudhury and Z. Yang, eds., *SVV*. 2006.
9. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, ed., *Formal Syntax and Semantics of Java*, *LNCS*, vol. 1523, pp. 41–82. Springer, 1999.
10. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In J. Alves-Foss, ed., *Formal Syntax and Semantics of Java*, *LNCS*, vol. 1523, pp. 241–269. Springer, 1999.
11. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security - Theories and Systems*, *LNCS*, vol. 3233, pp. 134–153. Springer, 2004.
12. K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, eds., *FM*, *LNCS*, vol. 3582, pp. 26–42. Springer, 2005.
13. C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, eds., *TPHOLs*, *LNCS*, vol. 3603. 2005.
14. T. F. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, eds., *Int. Workshop on the HOL Theorem Proving System and its Applications*, pp. 350–357. IEEE Computer Society Press, 1992.
15. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, eds., *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS*, vol. 1785, pp. 63–77. Springer, 2000.
16. O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
17. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, eds., *TPHOLs*, *LNCS*, vol. 1479, pp. 349–366. Springer, 1998.
18. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, *LNCS*, vol. 2283. Springer, 2002.
19. T. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe—definitely. In *ACM Symp. Principles of Programming Languages*, pp. 161–170. ACM Press, 1998.
20. Unified modeling language specification (version 1.5). Available as OMG document formal/03-03-01.
21. L. C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. Plotkin, C. Stirling, and M. Tofte, eds., *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pp. 187–211. MIT Press, 2000.
22. G. Smith, F. Kammüller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, eds., *ZB*, *LNCS*, vol. 2272, pp. 82–99. Springer, 2002.
23. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, eds., *FME*, *LNCS*, vol. 2391, pp. 89–105. Springer, 2002.