A Package for Extensible Object-Oriented Data Models with an Application to IMP++

Achim D. Brucker and Burkhart Wolff
{brucker, bwolff}@inf.ethz.ch

July 5, 2006

Information Security, ETH Zurich, CH-8092 Zurich, Switzerland

We present a datatype package that enables the use of shallow embedding technique to object-oriented specification and programming languages. The package incrementally compiles an object-oriented data model to a theory containing object-universes, constructors, and accessor 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 (OO) programming is a widely accepted programming paradigm, theorem proving over OO programs or OO 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 OO 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) *pre-compilation* into standard logic, and 2) *deep embeddings* into a meta-logic. As pre-compilation tools, for example, we consider Boogie for Spec# [2, 11] and several based on the Java Modeling Language (JML) such as Krakatoa [12]. The underlying idea is to compile OO 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. In particular, it is not possible to prove the soundness of a Hoare calculus with respect to the operational semantics, and, moreover, the soundness of the verification generator with respect to this calculus.

Among the tools based on deep embeddings, there is a sizable body of literature on formal models of Java-like languages (e.g., [7, 8, 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 [15] or, for HOL-Z [5]. These embeddings have been used for substantial applications [3]. 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 OO languages in HOL . In this technique, an expression E of type T in some OO language must be translated into some HOL-expression E' of HOL-type T'. The translation should preserve well-typedness in both ways, in particular the subtype relation. However, by "translation" we do not mean a simple one-to-one conversion; rather, the translation might use the OO type system for a pre-processing 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 OO 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 $oid \Rightarrow \mathscr{U}$ 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 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 clearly unfeasible.

As contributions of this paper, we 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 conservative definitions representing an OO 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 OO features and show the soundness of a Hoare calculus.

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. 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., $\alpha :: \mathit{order}$) 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 $\_ \wedge \_$, $\_ \vee \_$, $\_ \to \_$ of HOL have type $[\mathrm{bool}, \mathrm{bool}] \Rightarrow \mathrm{bool}$, $\neg\_$ has type bool $\Rightarrow$ bool. The quantifiers $\forall\, \_.\, \_$ and $\exists\, \_.\, \_$ have type $(\alpha \Rightarrow \mathrm{bool}) \Rightarrow \mathrm{bool}$. Quantifiers may range over higher order types, i.e., functions.

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

We assume a type class $\alpha :: \mathrm{bot}$ for all types α that provide an exceptional element $\perp$; for each type in this class a test for defindness is available via $\mathrm{def}\, x \equiv (x \neq \perp)$. The HOL type constructor $\tau_\perp$ assigns to each type τ a type *lifted* by $\perp$. Thus, each type $\alpha_\perp$ is member of the class bot. The function $\llcorner\_\lrcorner : \alpha \to \alpha_\perp$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\perp \to \alpha$ its inverse for defined values.

3 Typed Object Universes in an Object Store

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

Attributes with class types, e.g., the attribute `left` of class `Node`, are encoded using the type *oid*. These object identifiers (i.e., references) will be resolved by accessor functions like $A.\mathrm{left}$ for a given state; an access failure will be reported by $\perp$. Details of the accessor function definition process are described elsewhere [6].

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 OO type system), e.g., for class `Node` we assign an abstract datatype $\mathrm{Node_t}$ with the only element $\mathrm{Node_{key}}$. Further, for each class we introduce a base class type:

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

1. classes without attributes are represented by $(t \times \mathrm{unit}) \in \mathfrak{B}$, where $t \in \mathfrak{T}$ and unit is 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 $i \in \{0, \dots, n\}$ then $(t \times a_0 \times \cdots \times a_n) \in \mathfrak{B}$.

Thus, the base object type of class `Node` is $\mathrm{Node_t} \times \texttt{Integer} \times \mathit{oid} \times \mathit{oid}$ and of class `Cnode` is $\mathrm{Cnode_t} \times \texttt{Boolean}$.

Without loss of generality, we assume in our object model a common supertype of all objects. In the case of OCL (Object Constraint Language), 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 $\mathrm{Object_t} \in \mathfrak{T}$ be the tag of the common supertype `Object` and *oid* the type of the object identifiers,

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

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 OO semantics the non-referential setting is appropriate, where objects are viewed more like values. We discuss the consequences of this choice elsewhere in more detail [6]. 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 $\mathfrak{U}_{\mathrm{ref}}$ resp. $\mathfrak{U}_{\mathrm{nref}}$ (abbreviated $\mathfrak{U}_{\mathrm{x}}$) is inductively defined by:

1. $\mathscr{U}^0_\alpha \in \mathfrak{U}_{\mathrm{x}}$ is the initial universe type with one type variable (hole) α.

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

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

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

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 $C_0,\dots,C_{n-1}$ be the supertypes of C, i.e, C_i is inherited from C_{i-1}. The class type of C is defined as:
1. $C_i \in \mathfrak{B}, i \in \{0,\dots,n\}$ then
   $\mathscr{C}^0_\alpha = \Big(C_0 \times \Big(C_1 \times \big(C_2 \times \ldots \times (C_n \times \alpha_\perp)_\perp)_\perp\big)_\perp\Big)_\perp \in \mathfrak{C}$,
2. $\mathfrak{U}_\mathfrak{C} \supset \mathfrak{C}$, where $\mathfrak{U}_\mathfrak{C}$ is the set of universe types with $\mathscr{U}^0_\alpha = \mathscr{C}^0_\alpha$.

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

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

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. We see our definition as the more general one, since it allows for "partial objects" potentially relevant for other OO semantics for programming languages.

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.

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

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

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

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

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.

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:

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

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 $\bot$:

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

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

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

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.:

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

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 $\big((\alpha_1, \beta)\ \text{Node}\big)\ \texttt{Set}$) can also contain Cnodes. For resolving operation overloading, i.e., late-binding, the packages generates operation tables user-defined operations. This construction is described in [6].

4 The Package

Beside defining the presented definitions, the package proves that our encoding of object-structures is a faithful representation of OO (e.g., in the sense of language like Java or Smalltalk or the UML standard [20]). These theorems are proven for each class, e.g., during loading a specific UML 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. This includes properties of the object structure, e.g., that our conversion between universe representations and object representation is lossless, i.e., by proving:

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

In the rest of this section, we show how our package encodes recursive data structures with invariants and explain the underlying method.

4.1 Encoding Recursive Object Structures

A main contribution of our work is the encoding of *recursive* object structures, including the support for class invariants. First we introduce some basic notion: for arbitrary binary HOL operations *op*, we write $\tau \vDash P\ op\ Q$ for $\ulcorner P\ \tau \urcorner\ op\ \ulcorner Q\ \tau \urcorner$. Moreover, we write $\tau \vDash \partial\ x$ ("x is defined in state τ") for $\text{def}(x\ \tau)$, and $\tau \vDash \not\partial\ x$ for the contrary. We use generated accessor functions *self*.left and *self*.right that select a component in an object and de-reference the *oid* in the state τ.

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:

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

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 $\tau \vDash \mathit{self}\texttt{->IsType}(\mathrm{Node})$ (essentially a notation for the previously defined type tests), we can construct another approximation function (which has obviously the same type as NodeKindF):

$$\begin{aligned}\mathrm{NodeTypeF} \equiv \lambda\, \tau.\ \lambda\, X. \big\{\mathit{self}\ \big|\ &(\mathit{self} \in (\mathrm{NodeKindF}\ \tau\ X))\\ &\wedge \tau \vDash \mathit{self}\texttt{->IsType}(\mathrm{Node})\big\}\end{aligned}$$

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

$$\begin{aligned}&\mathrm{NodeKindSet} :: \mathscr{U}^1_{(\alpha_\mathrm{C},\beta_\mathrm{C},\beta_\mathrm{N})}\ \mathrm{St} \Rightarrow \mathscr{U}^1_{(\alpha_\mathrm{C},\beta_\mathrm{C},\beta_\mathrm{N})}\ \mathrm{St} \Rightarrow (\alpha_C, \beta_C)\ \texttt{Node}\ \mathrm{set}\\ &\mathrm{NodeKindSet} \equiv \lambda\, \tau.\, (\mathrm{gfp}(\mathrm{NodeKindF}\ \tau))\,.\end{aligned}$$

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

$$\begin{aligned}\tau \vDash \mathit{self} \in \mathrm{NodeKindSet} = \tau \vDash \partial\, \mathit{self}.\mathrm{i} \wedge{}& \tau \vDash \mathit{self}.\mathrm{i} > 5\\ \wedge\, \tau \vDash \partial\, \mathit{self}.\mathrm{left} \wedge{}& \tau \vDash (\mathit{self}.\mathrm{left}) \in \mathrm{NodeKindSet}\\ \wedge\, \tau \vDash \partial\, \mathit{self}.\mathrm{right} \wedge{}& \tau \vDash (\mathit{self}.\mathrm{right}) \in \mathrm{NodeKindSet}\end{aligned}$$

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

$$\mathrm{NodeTypeSet} \subseteq \mathrm{NodeKindSet}$$

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 coercion functions described in the previous section; we write $o_{[\mathrm{Node}]}$ 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:

$$\mathrm{CnodeF} \equiv \lambda\, \tau.\ \lambda\, X. \big\{\mathit{self}\ \big|\ \mathit{self}_{[\mathrm{Node}]} \in (\mathrm{NodeKindF}\ \tau\ (\lambda\, o.\, o_{[\mathrm{Node}]})\ \grave{}\ X)) \wedge \cdots\big\}$$

where the ... stand for the constraints specific to the subclass and $\grave{}$ denotes the pointwise application.

Similar to [4] 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 OO 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:

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

And also, 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:

$$\tau \vDash \mathit{self} \in \mathfrak{K}_C \longrightarrow \tau \vDash \mathit{self}_{[\mathrm{Node}]} \in \mathfrak{C}_N\,.$$

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.

4.2 The Underlying Method

Our OO 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 combined with a corresponding $\not\emptyset$-statement by disjunction, and
3. one allowing undefined references *and* undefined value types, i.e., all attribute accesses are combined with an corresponding $\not\emptyset$-statement by disjunction.

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 under 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. Overall, we believe that the support of different invariants is a corner stone of successful verification of OO systems, see also [11] where the authors arguing for weakened invariants, e.g., for inner calls for specifications written in JML.

5 Application: A Shallow Embedding of IMP++

In the following, we will interface the generated datatype theories of the package to a small, non-trivial OO language and show that compact calculi for denotational, operational and axiomatic semantics can be derived in a standard exercise. In particular, we show that proof-work as well as usability is fairly similar to previous work on IMP [24, 17], but notably easier than traditional work based on deep embeddings for OO languages since binding and typing are internalized. The small language follows deliberately the standard presentation of IMP in the Isabelle/HOL library inspired by a standard textbook on program semantics [24], but extends it with typedness, treatment of undefinedness, object-creation and object-update. In a small example, we sketch how to apply it for reasoning on weak and strong data invariants on tree-like structures.

There are essentially two ways to represent Boolean and value expressions in a shallow representation for IMP: either we try to represent them as functions from the HOL library directly. Or we use an own language of operations that take undefinedness into account and hides the λ-wiring of state passing away such as the HOL-OCL library [6]. The former has the advantage of direct reuse of Isabelle's powerful arithmetic decision procedures, while the latter has the advantage of representing "realistic" operational behavior: "$1 \operatorname{div} 0 = 1 \operatorname{div} 0$" is simply true in the former variant, but an exception in the latter. Since it is our main purpose to show applicability, i.e., proximity to specification languages such as JML, OCL, VDM or Spec#, we opt for the latter.

We re-interpret the logical judgments $\tau \vDash X \; op \, Y$ of the previous section to $\tau \vDash X \; (S op) \; Y = \llcorner \mathrm{true} \lrcorner$ and define $S op$ as the lifted strictified versions of op:

$$S \; op \; X \; Y = \lambda \tau. \begin{cases} \llcorner \ulcorner X \; \tau \urcorner \;\; op \;\; \ulcorner Y \; \tau \urcorner \lrcorner & \text{if } X \; \tau \neq \bot \wedge X \; \tau \neq \bot, \\ \bot & \text{otherwise}. \end{cases}$$

Thus, we generate context-lifted strictified versions for the basic operators $\_ \doteq \_$,$\_ \dot{\wedge} \_$, $\_ \dot{\vee} \_$, $\_ + \_$, $\_ * \_$, $\_ \cup \_$, $\_ \cap \_$, etc. Recall that the operations $\tau \vDash \partial \, x$ and $\tau \vDash \not\partial \, x$ are lifted versions of $\operatorname{def} x$ and $\neg \operatorname{def} x$; note, however, that they are non-strict and can be used to test if the access to program variables or references in the store has been successful.

5.1 Program Variables and Their Typed Semantic Interface.

Traditionally, program variables are just an object identifier (oid) representing the reference into the state, which is just a partial map $oid \rightharpoonup \alpha$. In a typed setting, we need also an embedding and projection pair and a sets denoting domain and co-domain of the functions between universe and class types. Thus, we represent a program variable by a tuple $(\alpha :: \mathrm{bot}, \beta :: \mathrm{bot})\mathtt{Var}$ with the components:

$$(\mathrm{ref} :: oid, \mathrm{proj} :: \alpha :: \mathrm{bot} \Rightarrow \beta, \mathrm{emb} :: \beta :: \mathrm{bot} \Rightarrow \alpha, \mathrm{dom} :: \alpha \ \mathrm{set}, \mathrm{cod} :: \alpha \ \mathrm{set})$$

satisfying the properties:

$$\begin{aligned} \forall x \in \operatorname{cod} v. \, \big((\operatorname{proj} v) \circ \big(\operatorname{emb} v\big)\big) \, x = x \wedge \forall x \in \operatorname{dom} v. \, ((\operatorname{emb} v) \circ (\operatorname{proj} v))x = x \\ \wedge(\operatorname{emb} v \, \bot) = \bot \wedge \forall x. \, \operatorname{def}(\operatorname{emb} v \, x) = (\operatorname{def} \, x) \\ \wedge(\operatorname{proj} v \, \bot) = \bot \wedge \forall x. \, \operatorname{def}(\operatorname{proj} v \, x) = (\operatorname{def} \, x) \end{aligned}$$

Variables are encapsulated in the type-constructor $(\alpha :: \mathrm{bot}, \beta :: \mathrm{bot})\mathtt{Var}$; they are instantiated by the package with the derived injection-projection pairs for class types.

The variable look-up operation $\#V$ has type $\alpha \ \mathrm{state} \Rightarrow \beta$. It is defined to yield $\bot$ for undefined references and a suitably converted value of type β otherwise. Due to the lifting, we can use updates directly inside expressions: $\tau \vDash \#X + 1 \doteq \#Y$ is a legally typed expression provided that X and Y are program variables with appropriate type interface. For example, $\mathrm{mk}_{\mathrm{Node}}$ and $\mathrm{get}_{\mathrm{Node}}$ are the instances for *emb* and *proj*, and *dom* and *cod* can be set to appropriate characteristic sets.

Note, however, that it is far too restrictive to chose a strict user-defined class invariant like NodeKindSet for this purpose. This would constrain the programming language, i.e., only object systems where all references in all objects must be defined at any time could be constructed. Therefore, it is sensible to chose a more liberal version of it allowing undefined references, i.e., version 2) described in section 4.2 is an appropriate candidate.

5.2 Syntax

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

$$\begin{aligned} \alpha \ \mathrm{com} = \ & \mathtt{SKIP} && \mid \alpha \ \mathrm{com} \ ; \alpha \ \mathrm{com} \\ & \mid \mathtt{CMD} \, \alpha \ \mathrm{cmd} && \mid \mathtt{IF} \ \alpha \ \mathrm{bexp} \ \mathtt{THEN} \ \alpha \ \mathrm{com} \ \mathtt{ELSE} \ \alpha \ \mathrm{com} \\ & \mid \mathtt{Assign} \ \ oid \ (\alpha \ \mathrm{aexp}) && \mid \mathtt{WHILE} \, \alpha \ \mathrm{bexp} \ \mathtt{DO} \ \alpha \ \mathrm{com} \end{aligned}$$

Here, α bexp is a synonym for $\alpha\ state \Rightarrow bool_\perp$ and cmd for $\alpha\ state \Rightarrow \alpha\ state_\perp$. In this definition, the assignment is untyped, i.e., the type of the variable is unrelated to the "intended type" of the *oid*. The following notation overcomes this problem:

$$X :== E \triangleq \texttt{Assign}(\mathrm{Oid}\ X)\big((\mathrm{Emb}\ Y) \circ E\big)$$

where Oid and Emb are the suitable projections and embeddings. Throughout this paper, we will assume that IMP++ assignments have this format.

5.3 Denotational Semantics

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

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

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

$$\begin{aligned} C(\texttt{SKIP}) &= \mathrm{Id} \\ C(\texttt{CMD}\ \ f) &= \{(s,t).\, s = \perp \wedge t = \perp\} \cup \{(s,t).\ \mathrm{def}\, s \wedge t = f\ulcorner s \urcorner\} \\ C(\texttt{Assign}\ oid\ a) &= \{(s,t).\, s = \perp \wedge t = \perp\} \\ & \quad \cup \{(s,t).\ \mathrm{def}\, s \wedge \neg\, \mathrm{def}(a\ulcorner s \urcorner) \wedge t = \perp\} \\ & \quad \cup \{(s,t).\ \mathrm{def}\, s \wedge \mathrm{def}(a\ulcorner s \urcorner) \wedge t = \llcorner \ulcorner s \urcorner(oid \mapsto a\ulcorner s \urcorner)\lrcorner\} \\ C(c_0; c_1) &= C(c_1) \circ_\perp C(c_0) \\ C(\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2) &= \{(s,t).\, (s = \perp \vee b\ulcorner s \urcorner = \perp) \wedge t = \perp\} \\ & \quad \cup \{(s,t).\ \mathrm{def}\, s \wedge \mathrm{def}(b\ulcorner s \urcorner) \wedge b\ulcorner s \urcorner = \llcorner \mathrm{true} \lrcorner \wedge (s,t) \in C c_1\} \\ & \quad \cup \{(s,t).\ \mathrm{def}\, s \wedge \mathrm{def}(b\ulcorner s \urcorner) \wedge b\ulcorner s \urcorner = \llcorner \mathrm{false} \lrcorner \wedge (s,t) \in C c_2\} \\ C(\texttt{WHILE}\ b\ \texttt{DO}\ c) &= \mathrm{lfp}(\Gamma\ b\ (C\ c)) \end{aligned}$$

where Γ is the usual approximation functional for the least fix-point operator lfp, enriched by the cases for undefined states. Based on C, the derivation of a natural semantics rules like:

$$\sigma \vDash \not\partial\, a \Longrightarrow \langle x{:==}a, \llcorner \sigma \lrcorner \rangle \longrightarrow_c \sigma' = (\sigma' = \perp)$$

is now a routine task. Thus, symbolic computations of programs is similarly done (and similarly efficiently) to IMP with the Isabelle simplifier.

5.4 Hoare Semantics

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

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

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

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

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

$$\frac{}{\models \{\lambda\, \sigma.\, \sigma \models \mathrm{err}\}\ c\ \{\lambda\, \sigma.\, \sigma \models \mathrm{err}\}} \qquad \frac{}{\models \{\odot \lambda\, \sigma.\, \ulcorner\sigma\urcorner \models \partial\, f \wedge Q(f\ulcorner\sigma\urcorner)\}\,\texttt{CMD}\, f\{\odot Q\}}$$

$$\frac{}{\models \{\odot \lambda\, \sigma.\, (\ulcorner\sigma\urcorner \models \partial\, a) \wedge Q(\llcorner\ulcorner\sigma\urcorner(oid \mapsto a\ulcorner\sigma\urcorner)\lrcorner)\}(\texttt{Assign}\ oid\ a)\{\odot Q\}}$$

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

Table 1: The Hoare Calculus for IMP++

detour via a derivability notion $\vdash$. We define a test for error states ($\mathrm{err} \equiv (\lambda\, x.\, \llcorner x = \bot\lrcorner)$ and write $\sigma \models \mathrm{err}$ in predicates. Moreover, we use the abbreviation $\odot\, P$ for $\lambda\, \sigma.\, \neg(\sigma \models \mathrm{err}) \wedge P\sigma$. Thus, assertions like $\models \{\odot\, P'\}c\{\odot\, Q'\}$ relate "legal" states. The derived calculus is then surprisingly standard (see Tab. 1).

5.5 An Example in IMP++.

The following IMP++program creates a cyclic data-structure over the class Node consisting of two nodes satisfying the invariant:

$$\begin{array}{l}\models\{\lambda\, \sigma.\, \sigma \models \partial\, \#H1 \wedge \sigma \models \partial\, \#H2\} \\ \quad \texttt{CMD New}(\#H1)(7); \texttt{CMD New}(\#H2)(9); \\ \quad \texttt{CMD Update}_{\texttt{left}}(\#H1)(\#H2); \texttt{CMD Update}_{\texttt{left}}(\#H2)(\#H1); \\ \quad \texttt{CMD Update}_{\texttt{right}}(\#H1)(\#H1); \texttt{CMD Update}_{\texttt{right}}(\#H2)(\#H2) \\ \ \{\lambda\, \sigma.\, \sigma \models \#H1.\,\texttt{content} \in \mathrm{Node} \wedge \sigma \models \#H2.\,\texttt{content} \in \mathrm{Node}\}\end{array}$$

To model local program variables of "reference type," we assume an extension of the original class system by a new class $\mathrm{Node}_{\mathrm{aux}}$ with a fixed attribute `content` of type Node. Further, we define the generator $\texttt{New}_{\texttt{content}}(Obj)(val)$ which generates a new Node object, initializes the data attribute with *val* (checking the liberal invariant), and stores the reference into the `content` field of *Obj*. Similarly, the operations $\texttt{Update}_{\texttt{left}}(Obj)(X)$ (and $\texttt{Update}_{\texttt{right}}(Obj)(X)$) updates the `left` attribute of the object given by the content attribute of *Obj*. All erroneous situations are reported by an error state. These operations can be generated automatically, including the necessary facts on the equalities of the projections into `left` and `right` attributes.[1]

The example 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. Recently, [11] suggested local flags in objects 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.

[1]At present, this is not supported by the package.

6 Conclusion

We presented an extensible universe construction supporting OO features such as subtyping and (single) inheritance. The construction is deeply intertwined with the concept of state. Class types are semantically explained via characteristic sets defined by greatest fixedpoints; these sets also give a semantics for class invariants. Various versions of invariants may be introduced and generated as well as their semantic relation be proven via co-induction. Thus, sufficient information may be generated in advance needed for proofs over programs building up object systems in an incremental way (locally neglecting global invariants that are used to described "consistent states" after a big step).

The universe-construction is supported by a package (developed as part of the HOL-OCL project [6]). Generated theories on object systems can be applied for OO programming language embeddings using the shallow technique. For such a programming language representation, a notable simplification is achieved both with respect to meta-theoretic reasoning (i.e., deriving calculi) as well as for efficient deduction.

One might object that the universe construction described in Sec. 3 and 4 is entirely meta-theoretic, thus not verifiable; and principles like conservative definitions are not applicable. However, while concepts like "the set of all HOL-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. In [6], an example suite of class diagrams is shown. The computation time for each of these models is below 2 minutes on recent hardware.

6.1 Related Work

Work on OO 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 [16]. 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 [13, 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.

Recently, Huffman et al [9] suggest a universe construction based on Scott's reflexive domains. Not really a package, merely a library construction, it helps to reflect the type constructor classes in Haskell-like languages.

The underlying encoding used by the loop tool [10] and Jive [14] 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.

6.2 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 languages like IMP++ is also desirable.
- *Support for Inductive Constraints.* By introducing measure-functions over object-structures, inductive datatypes can be characterized for defined measures of an object. This paves the way for the usual structural induction and well-founded recursion schemes,
- *Support of built-in Co-recursion.* Co-recursion can be used to define e.g., deep object equalities.
- *Deriving VCG.* Similar to the IMP-theory, verification condition generators for IMP++ programs can be proven sound and complete. This leads to effective program verification techniques based entirely on derived rules.

References

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, Orlando, 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 *Formal Methods*, *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, 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.

[6] A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zürich, 2006.

[7] 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.

[8] 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.

[9] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in isabelle/HOLCF. In J. Hurd and T. F. Melham, eds., *TPHOLs*, *LNCS*, vol. 3603, pp. 147–162. Springer, 2005.

[10] 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.

[11] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, eds., *Formal Methods*, *LNCS*, vol. 3582, pp. 26–42. Springer, 2005.

[12] 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.

[13] 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.

[14] 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.

[15] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[16] 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.

[17] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 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$_{\ell ight}$ is type-safe—definitely. In *ACM Symp. Principles of Programming Languages*, pp. 161–170. ACM Press, 1998.

[20] OMG Unified Modeling Language Specification. 2003.

[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., *Formal Methods*, *LNCS*, vol. 2391, pp. 89–105. Springer, 2002.

[24] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.