Using Theory Morphisms for Implementing Formal Methods Tools

Achim D. Brucker and Burkhart Wolff

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 52, D-79110 Freiburg, Germany
{brucker,wolff}@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/~{brucker,wolff}

Abstract Tools for a specification language can be implemented *directly* (by building a special purpose theorem prover) or *by a conservative embedding* into a typed meta-logic, which allows their safe and logically consistent implementation and the reuse of existing theorem prover engines. For being useful, the conservative extension approach must provide derivations for several thousand "folklore" theorems.
In this paper, we present an approach for deriving the mass of these theorems mechanically from an existing library of the meta-logic. The approach presupposes a structured *theory morphism* mapping library datatypes and library functions to new functions of the specification language while uniformly modifying some semantic properties; for example, new functions may have a different treatment of undefinedness compared to old ones.
Keywords: Formal Methods, Formal Semantics, Shallow Embeddings, Theorem Proving, OCL

1 Introduction

In contrast to a programming language, which defines computations, a *specification language* defines *properties* of computations, usually by extending a programming language with additional constructs such as quantifiers or universally quantified variables. Among the plethora of specification languages that has been developed, we will refer here only to examples such as Hoare-Logics [1, 2], Z [3, 4] or its semantic sister Higher-order Logics (HOL) [5], which has been advertised as "functional language with quantifiers" recently [6].

For the formal *analysis* of specification languages, their representation, i.e. their *embedding*, within a logical framework based on typed λ-calculi such as NuPRL [7], Coq [8] or Isabelle [9, 10] is a widely accepted technique that has been applied in many studies in recent years. With respect to *tools* implementing specification languages, the situation is not so clear-cut: while *direct* implementations in a programming environment are predominant [11, 12, 13], which result in special logic, special purpose theorem provers sometimes based on ad-hoc deduction technology, only a few tools are based on embeddings [14, 15, 16].

H. Geuvers and F. Wiedijk (Eds.): TYPES 2002, LNCS 2646, pp. 59–77, 2003.
© 2003 Springer-Verlag. This is the author's version of the work. It is posted at http://www.brucker.ch/bibliography/abstract/brucker.ea-embedding-2003 by permission of Springer-Verlag for your personal use. The definitive version was published with doi: 10.1007/3-540-39185-1_4.

There are two main advantages of the embedding approach: Beside the reuse of existing theorem prover engines, building such tools based on a *conservative* embedding into a logical framework also guarantees the safety and relative logical consistency of the tool. Unfortunately, in order to be practically useful *and* consistency-aware, the conservative embedding approach must provide derivations for several thousand "folklore theorems" (such as the associativity of the concatenation on lists or the commutativity of the union on sets) of the underlying logics or the basic datatypes of a specification language.

Based on the observation that in many language embeddings the bulk of function definitions follows a common scheme, our contribution in this paper consists of a method to structure these definitions into a modular theory morphism and a technique that exploits this structure and attempts to automatically derive "folklore theorems" from their counterparts in the meta-logic. Thus, upgraded libraries of the meta-logic can lead automatically to new theorems in the object logic since generic tactical support can "transform" theorems over functions of the meta-level into theorems at the object level. To say it loud and clear: we do not expect that *all* functions of a language semantics will be amenable to our approach; for the 10 percent that are core language constructs, we expect more or less standard verification work for properties of the language. But for the 90 percent that are library functions, our approach may significantly facilitate the embedding approach and lead to more portability.

This work was partly motivated by the development of *HOL-OCL* [17, 18] a conservative embedding of the Object Constraint Language (OCL) [19, 20, 21] into HOL. OCL is a textual extension of the object-oriented Unified Modeling Language (UML) [22] which is widely used within the object oriented software development process. In principle, OCL is a subtyped, three-valued Kleene-Logic with equality that allows for specifying constraints on graphs of object instances whose structure is described by UML class diagrams.

This paper proceeds as follows: after a presentation of the foundation of this work, we propose a structuring of the theory morphism into layers and present for each layer some typical combinators that capture the essence of semantic transformation from a meta-logical function to an object-logical one. We discuss the theory of these combinators and conceptually describe the tactics that perform the generation of generic theorems and the transformation of meta-level "folklore theorems" to their object-logical counterparts by means of a conservative theory morphism.

2 Foundations

In the following section, we will introduce a formal framework in order to define the core notion of "conservative theory morphism" which leads to the key observations and their practical consequences for the construction of language embeddings. The purpose of these abstract definitions is to demonstrate that our approach is in fact fairly general and applies to a wide range of proof systems based on higher-order typed calculi. In the subsequent sections, we present a

comparison of embedding techniques and introduce the underlying terminology of our approach. Finally, we outline the context of our running example.

2.1 Formal Preliminaries: The Generic Framework

In this section, we will introduce a formal framework in order to define the core notion of "conservative theory morphism" which leads to the key observations and their practical observations for the construction of shallow embeddings. The terminology used here follows the framework of *institutions* [23]. Throughout this paper, however, it is sufficient to base our notions on simple set-theoretic concepts instead of full-blown category theory. The concept of signature is inspired by [24], but can be expressed in other typed λ-calculi too.

First we introduce the notion of *sorts*, *types* and *terms*; we assume a set ρ of sorts and a set χ of *type constructors*, e.g. $\mathit{bool}, _ \rightarrow _, \mathit{list}, _\ \mathit{set}$. We assume a *type arity* ar, i.e. a finite mapping from type constructors to non-empty lists of sorts $ar : \chi \rightarrow_{\text{fin}} \mathit{list}_{\geq 1}(\rho)$. We define a set of types $\tau ::= \alpha \mid \chi(\tau, \ldots, \tau)$ based on the set of *polymorphic types* α. Further, we assume with $T(c, x)$ the *set of inductively defined terms* over constants c and variables x. For instance, for Isabelle-like systems, this set is defined as:

$$T(c, x) ::= c \mid x \mid T(c, x)T(c, x) \mid \lambda x.T(c, x)\ ,$$

while

$$\begin{aligned}
\chi &= \{_ \rightarrow _, \mathit{bool}\} && \text{(type constructors)}\\
\rho &= \{\mathit{term}\} && \text{(set of sorts)}\\
ar &= \{(\mathit{bool} \mapsto [\mathit{term}]), (_ \rightarrow _ \mapsto [\mathit{term}, \mathit{term}, \mathit{term}])\}. && \text{(type arity)}
\end{aligned}$$

A *signature* is a quadruple $\Sigma = (\rho, \chi, ar, c \rightarrow_{\text{fin}} \tau)$ and analogously the quadruple $\Gamma = (\rho, \chi, ar, x \rightarrow_{\text{fin}} \tau)$ is called an *environment*.

The following assumption incorporates a type inference and a notion of well-typed term: we assume a subset of terms called *typed terms* (written $T_{\Sigma,\Gamma}(c, x)$) and a subset *typed formulae* (written $F_{\Sigma,\Gamma}(c, x)$); we require that in these notions, ar, ρ and χ agree in Σ and Γ. For example, a *type inference system* for order-sorted polymorphic terms, can be found in [24]. Formulae, for example, can be typed terms of type *bool*.

We call $S = (\Sigma, A)$ with the axioms $A \subseteq F_{\Sigma,\Gamma}(c, x)$ a *specification*. The following assumption incorporates an inference system: with a *theory* $\mathit{Th}(S) \subseteq F_{\Sigma,\Gamma}(c, x)$ we denote the set of formulae derivable from A; in particular, we require $A \in \mathit{Th}(S)$ and Th to be monotonous in the axioms, i.e. $S \subseteq S' \implies \mathit{Th}(S) \subseteq \mathit{Th}(S')$ (we also use $S \subseteq S'$ for the extension of subsets on tuples for component-wise set inclusion).

A *signature morphism* is a mapping $\Sigma \rightarrow \Sigma$ which can be naturally extended to a *specification morphism* and a *theory morphism*.

The following specification extensions $S \subseteq S'$, called *conservative* specification extensions (see [5]), are of particular interest for this paper:

1. *type synonyms*,
2. *constant definitions*, and
3. *type definitions*.

A type synonym introduces a type abbreviation and is denoted as:

$$S' = S \uplus [\textbf{types}\ t(\alpha_1, \ldots, \alpha_n) = T(\alpha_1, \ldots, \alpha_n, t')] \ .$$

It is purely syntactical (i.e. it we will be used for abbreviations in type annotations only) such that the extension is defined by $S' = S$.

A constant definition is denoted as:

$$S' = S \uplus [\textbf{constdefs}\ \text{“}c = E\text{”}] \ .$$

A constant definition is *conservative*, if the following syntactic conditions hold: $c \notin \mathrm{dom}(\Sigma)$, E is closed and does not contain c, and no sub-term of E has a type containing a type variable, that is not contained in the type of c. Then S' is defined by $((\rho, \chi, ar, C'), A')$, where $S = ((\rho, \chi, ar, C), A)$ and $A' = A \cup \{c = E\}$ and $C' = C \cup \{(c \mapsto \tau)\}$ where τ is the type of E.

A type definition will be denoted as follows:

$$S' = S \uplus [\textbf{typedef}\ \text{“}T(\alpha_1, \ldots, \alpha_n) = \{x \mid P(x)\}\text{”}] \ .$$

In this case, $S' = ((\rho, \chi', ar', C'), A')$ is defined as follows: We assume $S = ((\rho, \chi, ar, C), A)$, and $P(x)$ of type $P :: R \to \mathit{bool}$ for a base type R in χ. C' is constructed from C by adding $\mathit{Abs}_T : R \to T$ and $\mathit{Rep}_T : T \to R$. χ' is constructed from χ by adding the new type T (i.e. which is supposed to be not in χ). The axioms A' is constructed by adding the two isomorphism axioms

$$A' = A \cup \{\forall x. \mathit{Abs}_T(\mathit{Rep}_T(x)) = x, \forall x. P(x) \implies \mathit{Rep}_T(\mathit{Abs}_T(x)) = x\} \ .$$

The type definition is conservative if the proof obligation $\exists x. P(x)$, holds.

Instead of $S \uplus E_1 \uplus \cdots \uplus E_n$ we write $S \uplus E$. Technically, conservative language embeddings are represented as *specification increments* E, that contain the type definitions and constant definitions for the language elements and give a semantics in terms of a specification S.

The overall situation is summarized in the following commutative diagram:

$$\begin{array}{ccc} S & \xrightarrow{\quad Th \quad} & Th(S) \\ \Big\downarrow{\scriptstyle \uplus E} & & E^{-1}\Big\uparrow \ \Big\downarrow \ \Big\Downarrow TM_E \\ S \uplus E & \xrightarrow{\quad Th \quad} & Th(S \uplus E) \end{array}$$

The three morphisms on the right of the diagram require some explanation: The injection $(\hookrightarrow)$ from $Th(S)$ to $Th(S \uplus E)$ is a consequence of the fact that $\uplus$

constructs extensions and *Th* is required to be monotonous. The theory morphism E^{-1} exists, since our extensions are *conservative*: all new theorems can be retranslated into old ones, which implies that the new theory is consistent whenever the old was (see [5] for the proof). The theory morphism TM_E (denoted by $\Rightarrow$) connects the $Th(S)$ to $Th(S \uplus E)$ and serves as specification for the overall goal of this paper, namely the construction of a partial function $LIFT_E : Th(S) \to TM_E(TH(S'))$ that approximates the functor TM_E.

Our Framework and Isabelle/HOL. Our chosen meta-logic and implementation platform Isabelle/HOL is the instance of the generic theorem prover Isabelle [10] with higher-order logic (*HOL*) [25, 26]. Isabelle directly implements order sorted types ([24]; Note, however, that we do not make use of the ordering on sorts throughout this paper), and supports the conservative extension schemes abstractly presented above. Isabelle/HOL is the instance of Isabelle that is most sophisticated with respect to proof-support and has a library of conservative theories. Among others, the HOL-core provides type *bool*, the number theories provide *nat* and *int*, the typed set theory provides *set*(τ) and the list specification provides *list*(τ). Moreover, there are products, maps, and even a specification on real numbers and non-standard analysis. The HOL-library provides several thousand theorems — yielding the potential for reuse in a specialized tool for a particular formal method.

Our Framework in the Light of other Type Systems. It is straightforward to represent our framework in type systems that allow *types depending on types* [27], i.e. the four λ-calculi on the backside of Barendregt's cube. In the weakest of these systems, $\lambda\underline{\omega}$, the same notion of sorts is introduced as in our framework. For example, the sort $*$ in $\lambda\underline{\omega}$ corresponds to *term*. The arities correspond to *kinds*, which are limited to $*$ in $\lambda\underline{\omega}$, however, since kinds are defined recursively by $\mathbb{K} = *|\mathbb{K} \to \mathbb{K}$, there are *higher-order* type constructors in $\lambda\underline{\omega}$ that have no correspondence in our framework. The arities of type constructors can be encoded by kinds: the arity for $_ \to _$, namely $[term, term, term]$ corresponds to the kind $* \to * \to *$. Declarations of type synonyms **types** $t(\alpha_1, \ldots, \alpha_n) = T$ correspond to $\lambda\alpha_1 : *, \ldots, \alpha_n : *.T$, etc.

2.2 Embedding Techniques — An Overview

For our approach, it is necessary to study the technique of embeddings realized in a theory morphism in more detail. While these underlying techniques are known since the invention of typed λ-calculi (see for the special case of the quantifiers in [25]), it was not before the late seventies that the overall importance of *higher-order abstract syntax* (a term coined by [28]) for the representation of binding in logical rules and program transformations [29] and for implementations [28] was recognized. The term "shallow embedding" (invented in [30]) extends higher-order abstract syntax (HOAS) to a semantic definition and is contrasted to "deep embeddings". Moreover, throughout this paper, we will distinguish *typed*

and *untyped* shallow embeddings. Conceptually, these three techniques can be summarized as follows:

Deep embeddings represent the abstract syntax as a datatype; variables and constants are thus represented as constants in the meta-logic. A semantics is defined "over" the datatype using a transition relation $\rightarrow_r$ or an interpretation function *Sem* from syntax to semantics.

Untyped shallow embeddings use HOAS to represent the syntax of a language by declaring uninterpreted constant symbols for all constructs *except* variables which are directly represented by variables of the meta-logic; thus, binding and substitution are "internalized" on the meta-level, but not the typing. A semantics is defined similarly to a deep embedding.

Typed shallow embeddings use HOAS but include also the type system of the language in the sense that ill-typed expressions can not be encoded well-typed into the meta-logic. This paves the way for defining the semantics of the language constructs and its functions by a direct definition in terms of the meta-logic, i.e. its theories for e.g. orders, sets, pairs, and lists.

The difference between these techniques and their decreasing "representational distance" is best explained by the simplest example of a typed language: the simple typed λ-calculus itself. The syntax can be declared as follows:

	Deep	Untyped Shallow	Typed Shallow
VAR:	$\alpha \rightarrow \mathrm{L}(\alpha, \beta)$		
CON:	$\beta \rightarrow \mathrm{L}(\alpha, \beta)$	$\beta \rightarrow \mathrm{L}(\beta)$	
LAM:	$\alpha \times \mathrm{L}(\alpha, \beta) \rightarrow \mathrm{L}(\alpha, \beta)$	$(\mathrm{L}(\beta) \rightarrow \mathrm{L}(\beta)) \rightarrow \mathrm{L}(\beta)$	$\mathrm{L}(\gamma,\delta) \rightarrow \mathrm{L}(\gamma,\delta)$
APP:	$\mathrm{L}(\alpha, \beta) \times \mathrm{L}(\alpha, \beta) \rightarrow \mathrm{L}(\alpha, \beta)$	$\mathrm{L}(\beta) \times \mathrm{L}(\beta) \rightarrow \mathrm{L}(\beta)$	$\mathrm{L}(\gamma,\delta) \times \gamma \rightarrow \delta$

where the underlying types can be defined by the equations:

Deep	Untyped Shallow	Typed Shallow
$L(\alpha, \beta) = \alpha \mid \beta$	$L(\beta) = \beta$	
$\mid \alpha \times L(\alpha, \beta)$	$\mid L(\beta) \rightarrow \times L(\beta)$	$L(\gamma, \delta) = \gamma \rightarrow \delta$
$\mid L(\alpha, \beta) \times L(\alpha, \beta)$	$\mid L(\beta) \times L(\beta)$	

The first type equation can be directly interpreted as a datatype and is thus inductive, the second can interpreted as datatype only with difficulties (requiring reflexive Scott Domains), while the third has clearly no inductive structure at all. Since the typed shallow embedding "implements" binding and typing efficiently by the meta-level, it is more suited for tool implementations. However, induction schemes over the syntax usually yield the crucial weapon for completeness proofs in various logics, for instance, and motivate therefore the use of deep embeddings in meta-theoretic reasoning.

To complete, we compare now the definition of semantics in all three settings:

Deep	Untyped Shallow	Typed Shallow
APP(LAM(x,F),A) $\rightarrow_\beta$	APP(LAM(F),A) $\rightarrow_\beta$F A	APP(F,A) = F A
subst(*alfa*(F,*free*(A)),x,A)		LAM(F) = F
+ congruence rules	+ congruence rules	

where $\rightarrow_\beta$ is just the usual inductively defined β-reduction relation, *subst* and *free* the usual term functions for substitution and computation of free variables and *alfa* is assumed to compute an α-equivalent term whose bound variables are disjoint from *free*(A). In an untyped shallow setting, these functions are not needed since variables and substitution are internalized into the meta-language. In the typed shallow embedding, APP is *semantically* represented by the application of the meta-language and LAM by the identity; the β-reduction APP(LAM(F),A) = F A is just a derived equality in the meta-logic. In a meta-logic assuming Leipnitz' Law for equality (such as HOL), congruence rules are not needed since equality is a universal congruence.

Note that the mapping in our typed shallow embedding between language and meta-language must not be so trivial as it is in this example; it can involve exception handling, special evaluation strategies such as call by value, backtracking, etc. Moreover, the relation between the type systems of the two languages may also be highly non-trivial. This is what our running example OCL will do in the next chapters.

Further, note the technical overhead between deep and shallow embeddings will even be worse if we introduce function symbols such as + and numbers 0,1,2, ... into our language. In the deep embedding, the whole syntax and semantics must be encoded into new datatypes and reduction relations over them, while in the typed shallow embedding, the operators of the meta-logic (possibly adapted semantically) can be reused more or less directly.

Summing up, a deep embedding on the one end of the spectrum requires a lot of machinery for binding, substitution and typing, while a the other end, binding and typing are internalized into the meta-logic, paving the way for efficient implementations using directly the built-in machinery of the theorem prover. Therefore, whenever we speak of an embedding in the sequel, we will assume a typed shallow embedding.

2.3 OCL in a Nutshell

The Unified Modeling Language (UML) is a diagrammatic specification language for modeling object oriented software systems. UML is defined in an open standardization process lead by the Object Management Group (OMG) and highly accepted in industry. Being specialized for the object-oriented software development process, UML allows to specify object-oriented data models (via class diagrams), using data encapsulation, *subtyping* (inheritance), *recursion* (in datatypes and function definitions) and *polymorphism* (overwriting).

While UML as a whole can only claim to be a semi-formal language, UML class-diagrams can be completed by the *Object Constraint Language* (OCL) to a (fully) formal specification language. A prominent use of OCL in [19] is the specification of class invariants and pre and post conditions of methods, e.g.:

```
context Account
  inv: Account.allInstances->forAll(a1,a2 |
                                    a1 <> a2 implies a1.id <> a2.id)
```

```
context Account::makeWithdrawal(amount:Real)
  pre:  (amount > 0) and (balance − mount) >= 0
  post: balance = balance@pre − amount
        and currency = currency@pre
```

The first example requires, that the attribute id of the class **Account** is unique for all instances in a given system state. The second example shows a simple pre/post condition pair, describing a method for withdrawal on an **Account** object. Note, that within post conditions one can access the previous state by using the **@pre**-keyword.

Being a typed logic that supports reasoning over object-graphs defined by object-oriented class diagrams, OCL reasons over path expressions of the underlying class diagram. Any path can be undefined *in a given state*; thus, the undefinedness is inherent in OCL.

3 Organizing Theory Morphisms into Layers

In practice, language definitions follow a general principle or a common scheme. In OCL, for example, there is the following requirement for functions except the explicitly mentioned logical connectors (_ **and** _, _ **or** _, **not** _) and the logical equality (_ = _):

> Object Constraint Language Specification [19] (version 1.4), page 6–58
>
> Whenever an OCL-expression is being evaluated, there is the possibility that one or more queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

In more standard terminology, one could rephrase this semantic principle as "all operations are strict", which is a special principle describing the handling of exceptions[1]. Further semantic principles are, for example, "all collection types are smashed" (see below), or, principles related to the embedding technique.

Instead of leaving these principles implicit inside a large collection of definitions, the idea is to capture their essence in *combinators* and to make these principles in these definitions explicit. Such combinators occur both on the level of types in form of type constructors and on terms in form of constant symbols.

As such, this approach is by no means new; for example, for some semantic aspects like exception handling or state propagation, *monads* have been proposed as a flexible means for describing the semantics of a language "facet by facet" in a modular way [31, 32]. While we will not use monads in this work (which is a result of our chosen standard example, OCL, and thus accidental), and while we do not even suggest a similar fixed semantic framework here, merely a discipline to capture these principles uniformly in combinators (may they have monad structure or not), we will focus on the potential of such a discipline, namely to express their theory once and for all and to exploit it in tactical programs.

[1] In this view, the logical equality can be used to "catch exceptions".

We turn now to the layering of our theory morphism. We say that a theory morphism is layered, iff in each form of conservative extension the following decomposition is possible:

$$\begin{array}{ll} \textbf{types} & \text{“}T('a_1 \ldots ' a_m) = C_n(\cdots (C_1(T'))\text{”} \\ \textbf{typedef} & \text{“}T(\alpha_1, \ldots, \alpha_m) = \{x :: C_n(\cdots (C_1(T')) \mid P(x)\}\text{”} \\ \textbf{constdefs} & \text{“}c = (E_n \circ \cdots \circ E_1)(c')\text{”} \end{array}$$

where each C_i or, respectively, E_i are (type constructor) expressions build from *semantic combinators* of *layer* S_i and T' respectively. Note, that c' is a construct from the meta logic. A *layer* S_i is represented by a specification defining the *semantic combinators*, i.e. constructs that perform the semantic transformation from meta-level definitions to object-level definitions. In Fig. 1, we present a classification for such layers.

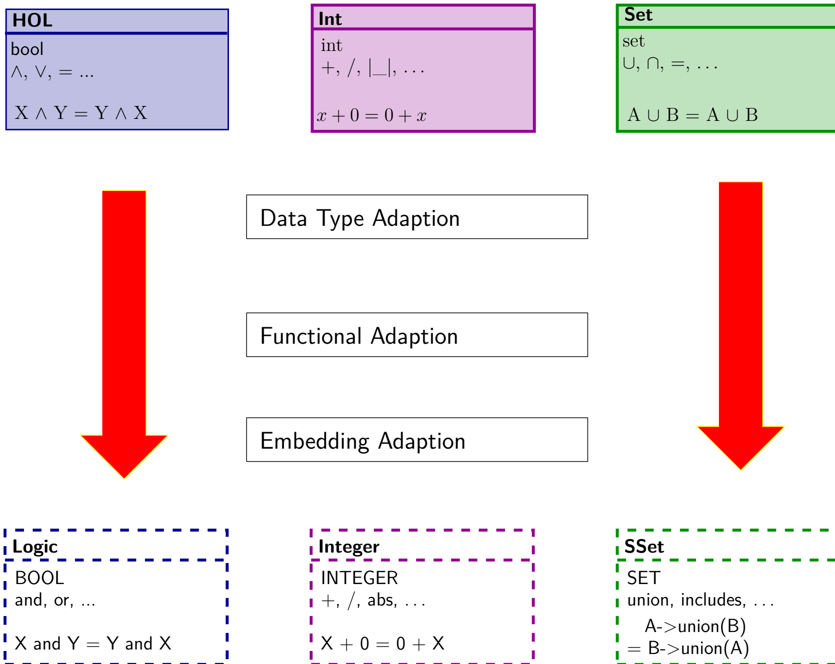

Figure 1. Derivation of the OCL-library

In the following sections, we will present a typical collection of layers and their combinators. We will introduce the semantic combinators one by one and collect them in a distinguished variable SEMCOM. Finally, we will put them together for our example OCL and describe generic theorem proving techniques that exploit the layering of the theory morphism for OCL.

3.1 Datatype Adaption

Datatype adaption establishes the link between meta-level types and object-level types and meta-level constants to object-level constants. While meta-level definitions in libraries of existing theorem prover systems are geared toward good tool support, object-level definitions tend to be geared to a particular computational model, such that the gap between these two has to be bridged. For example, in Isabelle/HOL, the *head*-function applied to an empty list is defined to yield an arbitrary but fixed element; in a typical executable object-language such as SML, Haskell or OCL, however, this function should be defined to yield an exception element that is treated particularly. Thus, datatype adaption copes with such failure elements, the introduction of boundaries (as maximal and minimal numbers in machine arithmetics), congruences on raw data (such as *smashing*; see below) and the introduction of additional semantic structure on a type such as complete partial orders (cpo).

We chose the latter as first example for a datatype adaption. We begin with the introduction of a "simple cpo" structure via the specification extension by sort *cpo*0 and the definition of our first semantic (type) combinator; simple cpo means that we just disjointly add a failure-element such as $\perp$ (see, e.g. [1], where the following construction is also called "lifting"). Note, that an extension to full-blown cpo's would require the additional definition of the usual partial definedness-ordering with $\perp$ as least element and completeness requirements; such an extension is straight-forward and useful to give some recursive constructs in OCL a semantics but out of the scope of this paper.

We state:

$$\textbf{datatype}\ \mathrm{up}(\alpha\ ) = \texttt{"}\lfloor(_)\rfloor\texttt{"}\ \alpha \mid \perp$$

which is a syntactic notation for a type definition and two constant definitions for the injections into the sum-type. In the sequel, we write $t_\perp$ instead of $up(t)$. For example, we can define the object-level type synonym *Bool* based on this combinator:

$$\textbf{types}\ \mathit{Bool} = \mathit{bool}_\perp \qquad \textbf{types}\ \mathit{Integer} = \mathit{integer}_\perp \qquad \ldots$$

These type abbreviations reflect the effect of the datatype adaption.

We turn now to the semantical combinators of this layer. We define the inverse to $\lfloor_\rfloor$ as $\lceil_\rceil$. We have defined a small specification extension providing the semantic combinators: $(_)_\perp, \perp, \lfloor_\rfloor, \lceil_\rceil \in \mathrm{SEMCOM}$.

As an example for a congruence construction, we chose *smashing on sets*, which occurs in the semantics of SML or OCL, for example. In a language with semantic domains providing $\perp$-elements, the question arises how they are treated in type constructors like product, sum, list or sets. Two extremes are known in the literature; for products, for example, we can have:

$$(\perp, X) \neq \perp \qquad \{a, \perp, b\} \neq \perp \qquad \ldots$$

or:

$$(\bot, X) = \bot \qquad \{a, \bot, b\} = \bot \qquad \ldots$$

The latter variant is called *smashed product* and *smashed set*. In our framework, we define a semantic combinator for smashing as follows:

constdefs smash :: $[[\beta$:: cpo0, α :: cpo0] $\to$bool, α] $\to\alpha$
"smash f X $\equiv$ if f $\bot$X then $\bot$else X"

and define, for example, *Set*'s as follows:

typedef α Set = "{X:: (α :: cpo0) set up.(smash (λx X. x : $\lceil$X$\rceil$) X) = X}"

An embedding of smashed sets into "simple cpo's" can be done as follows:

instance Set :: ord(term)
arities Set :: cpo0(term)
constdefs UU_Set_def "$\bot$ $\equiv$Abs$_{Set}$$\bot$"

We have defined the semantic combinators smash, $\bot :: Set(\alpha), \mathrm{Abs}_{Set}, \mathrm{Rep}_{Set} \in$ SEMCOM.

3.2 Functional Adaption

Functional adaption is concerned with the semantic transformation of a meta-level function into an object-level function. For example, this may involve the

- *strictification* of functions, i.e. the result of the function is undefined if one of its arguments is undefined,
- *late-binding-conversion* of a function. This semantic conversion process is necessary for converting a function into an function in an object-oriented language.

Technically, strictification can be achieved by the definition of the semantic combinators. We will introduce two versions: a general one on the type class *cpo*0, another one for the important variant:

constdefs
strictify :: "($\alpha_\bot$ $\to\beta$:: cpo0) $\to\alpha_\bot\to\beta$"
"strictify f x $\equiv$ if x=$\bot$ then $\bot$else f x"
strictify' :: "($\alpha_\bot$ $\to\beta$:: cpo0) $\to\alpha_\bot\to\beta$"
"strictify' f x $\equiv$ case x of $\lfloor$v$\rfloor$ $\to$(f v) | $\bot\to\bot$"

(strictify', strictify $\in$ SEMCOM).

A definition like OCL's union (that is the strictified version of HOL's union over the smashed and transformed HOL datatype *set*) is therefore represented as:

constdefs
union :: Set(α) $\to$Set(α) $\to$Set(α)
"union $\equiv$strictify(λ X. strictify(λ Y. Abs$_{Set}$$\lfloor\lceil$Rep$_{Set}X\rceil$ $\cup\lceil$Rep$_{Set}$Y$\rceil\rfloor$))"

Many object-oriented languages provide a particular call-scheme for functions, called *method invocation* which is believed to increase the reusabilty of code. Method invocation is implemented by a well-known construction in programming language theory called *late-binding*. In order to demonstrate the flexibility of our framework, we show in the following example how this important construction can be integrated and expressed as a semantic combinator. The late-binding-conversion requires a particular pre-compilation step that is not semantically treated by combinators: For each method declaration

Method m: $t_1, \ldots, t_n \to t$

in a class-declaration A, a look-up table lookup_m has to be declared with type:

$\text{lookup}_m :: \text{set}(A) \to A \to t_1 \times \ldots \times t_n \to t$

In an "invocation" $A.m(a_1, \ldots, a_n)$ of a "method of object A", the dynamic type of A is detected, which is used to lookup the concrete function in the table, that is executed with A as first argument (together with the other arguments). The dynamic type of a "class of objects A" can be represented by *set*[2]. Thus, the semantics of method invocations can be given by the following semantic combinators:

match lookup obj ≡the (lookup (LEAST X: α . X : dom lookup ∧obj: X)))
methodify lookup obj arg ≡(match lookup obj)(arg)

where we use predefined Isabelle/HOL functions for "the", "dom" and "LEAST" with the 'obvious' meaning. Since OCL possesses subtyping but *not* late-binding at the moment, we will not apply these combinators throughout this paper. The discussion above serves only for the demonstration that late-binding can in fact be modeled in our framework. A detailed account on the handling of subtyping can be found in [17].

3.3 Embedding Adaption for Shallow Embedding

This type of semantic combinators is related to the embedding technique itself. Recalling section 2.2, any function $op : T_1 \to T_2$ of the object-language has to be transformed to a function:

$$Sem_\sigma[\![op]\!] : V_\sigma(T_1) \to V_\sigma(T_2) \text{ where types } V_\sigma(\delta) = \sigma \to \delta\ .$$

The transformation is motivated by the usual form of a semantic definition for an operator op and an expression e in a deep embedding:

$$Sem_\sigma[\![op\, e]\!] = \lambda\sigma.(Sem_\sigma[\![op]\!]\sigma)(Sem_\sigma[\![e]\!]\sigma)$$

for some environment or state σ. Consequently, the semantics of an expression e of type T is given by a function $\sigma \to T$ (written as $V_\sigma(T)$). In a typed shallow

[2] This requires a construction of a "universe of objects" closed under subtypes generated by inheritance; in [17], such a construction can be found.

embedding, the language is constructed directly without the detour of the concrete syntax and *Sem*. Hence, all expressions are converted to functions from their environment to their value in T, which implies that whenever a language operators is applied to some arguments, the environment must be passed to them accordingly. This "plumbing" with the environment parameter σ is done by the semantic combinators K, lift_1 or $\text{lift}_2 \in$ SEMCOM that do the trick for constants, unary or binary functions. They are defined as follows:

$$\begin{array}{lll}
\text{K} & :: & \alpha \to V_\sigma(\alpha) \\
\text{"K a} & \equiv & (\lambda \text{st. a})\text{"} \\
\text{lift}_1 & :: & (\alpha \to \beta) \to V_\sigma(\alpha) \to V_\sigma(\beta) \\
\text{"lift}_1\ \text{f X} & \equiv & (\lambda \text{st. f (X st)})\text{"} \\
\text{lift}_2 & :: & ([\alpha, \beta] \to \gamma) \to [V_\sigma(\alpha), V_\sigma(\beta)] \to V_\sigma(\gamma) \\
\text{"lift}_2\ \text{f X Y} & \equiv & (\lambda \text{st. f (X st)(Y st)})\text{"}
\end{array}$$

Our "layered approach" becomes particularly visible for the example of the logical absurdity or the the logical negation operator (standing for similar unary operators):

$$\begin{array}{ll}
\textbf{constdefs} & \bot_{\mathscr{L}} :: V_\sigma(\text{Bool}) \\
& \text{"}\bot_{\mathscr{L}} \equiv \text{K}(\lfloor \bot \rfloor)\text{"} \\
& \mathsf{true} :: V_\sigma(\text{Bool}) \\
& \text{"}\mathsf{true} \equiv \text{K}(\lfloor \text{true} \rfloor)\text{"} \\
& \mathsf{false} :: V_\sigma(\text{Bool}) \\
& \text{"}\mathsf{false} \equiv \text{K}(\lfloor \text{false} \rfloor)\text{"} \\
& \\
& \text{not} :: V_\sigma(\text{Bool}) \to V_\sigma(\text{Bool}) \\
& \text{"not} \equiv (\text{lift}_1 \circ \lfloor _ \rfloor \circ \text{strictify'})\ (\neg)\text{"}
\end{array}$$

From this definition, the usual logical laws for a strict negation can be derived:

$$\mathsf{not}(\bot_{\mathscr{L}}) = \bot_{\mathscr{L}} \qquad \mathsf{not}(\mathsf{true}) = \mathsf{false} \qquad \mathsf{not}(\mathsf{false}) = \mathsf{true}$$

As an example for a binary function like Union (based on union defined in the previous section), we present its definition:

$$\begin{array}{lll}
\textbf{constdefs}\ \text{Union} & :: & V_\sigma(\text{Set}(\alpha)) \to V_\sigma(\text{Set}(\alpha)) \to V_\sigma(\text{Set}(\alpha)) \\
\text{Union} & \equiv & \text{lift}_2\ \text{union}
\end{array}$$

We will write BOOL for $V_\sigma(\text{Bool})$, INTEGER for $V_\sigma(\text{Integer})$ and SET(α) in the sequel. These type abbreviations reflect the effect of the embedding adaption on types.

4 Automatic Generation of Library Theorems

We distinguished two ways to generate theorems for newly embedded operators of an object language: instantiations from generic theorems over the semantic combinators or the application of LIFT_E, a tactic procedure that attempts to reconstruct meta-level theorems on the object-level.

4.1 Generic Theorems

In our example application OCL, definedness is a crucial issue that has been coped with by semantic combinators. Definedness is handled by the predicate is_def: $V_\sigma(\alpha) \to$BOOL that lifts the predicate DEF t $\equiv$(t $\neq\perp$) to the level of the OCL logic. Since the latter "implanted" undefinedness on top of the meta-level semantics, it is not surprising that there are a number of properties that are valid for all functions that are defined accordingly to the previous sections.

$$\begin{array}{l}
\text{is_def}(\text{lift}_1\,(\text{strictify'}(\lambda x.\ \lfloor f\ x\rfloor))\ X) = \text{is_def}\ X \\
\text{is_def}(\text{lift}_1\,(\text{strictify'}(\lambda x.\ \lfloor f\ x\rfloor))\ X) = \text{is_def}\ X \\
\text{is_def}(\text{lift}_2\,(\text{strictify'}(\lambda x. \\
\qquad\qquad \text{strictify'}(\lambda y.\ \lfloor f\ x\ y\rfloor)))\ X\ Y) = (\text{is_def}\ X \text{ and } \text{is_def}\ Y)
\end{array}$$

$$\begin{array}{l}
\text{lift}_1\,(\text{strictify'}\ f)\ \perp_{\mathscr{L}} = \perp_{\mathscr{L}} \\
\text{lift}_2\,(\text{strictify'}(\lambda x.\ \text{strictify'}(f\ x)))\ \perp_{\mathscr{L}} X = \perp_{\mathscr{L}} \\
\text{lift}_2\,(\text{strictify'}(\lambda x.\ \text{strictify'}(f\ x)))\ X\ \perp_{\mathscr{L}} = \perp_{\mathscr{L}}
\end{array}$$

For any binary function defined in the prescribed scheme, these theorems already result in four theorems simply by instantiating f appropriately!

Surprisingly, the embedding adaption combinators K, lift_1 and lift_2 turn out to have a quite rich theory of their own. First, it is possible to characterize the "shallowness" of a context C in the sense that the environment/store is just "passed through" this context. This characterization can be formulated semantically and looks as follows:

$$\begin{array}{lll}
\textbf{constdefs}\ \text{pass} & :: & ([V_\sigma(\gamma),\sigma] \to \beta) \to \text{bool} \\
\qquad\qquad \text{pass}(C) & \equiv & (\exists f.\ \forall\ X\ st.\ C\ X\ st = f\ (X\ st)\ st)
\end{array}$$

This predicate enjoys a number of useful properties that allow for the decomposition of a larger context C to smaller ones; for instance, trivial contexts pass and passing is compositional:

$$\begin{array}{ll}
\text{pass}(\lambda X.\ c) & \text{pass}(\lambda X.\ X) \\
[\![\ \text{pass } P;\ \text{pass } P'\ ]\!] & \Rightarrow \text{pass}(P \circ P')
\end{array}$$

Moreover, any function following the prescribed scheme is shallow (since this was the very reason for introducing the pass-predicate):

$$\begin{array}{ll}
[\![\ \text{pass } P\ ]\!] & \Rightarrow \text{pass}(\lambda X.\ \text{lift}_1\ f\ (P\ X)) \\
[\![\ \text{pass } P;\ \text{pass } P'\ ]\!] & \Rightarrow \text{pass}(\lambda X.\ \text{lift}_2\ f\ (P\ X)\ (P'\ X))"
\end{array}$$

This leads to a side-calculus enabling powerful logical rules like trichotomy (for the language composed by the operators):

$$\begin{array}{l}
[\![\ \text{pass } P;\ \text{pass } P';\ P\ \perp_{\mathscr{L}} = P'\ \perp_{\mathscr{L}};\ P\ \mathsf{true} = P'\ \mathsf{true};\ P\ \mathsf{false} = P'\ \mathsf{false}]\!] \\
\quad \Rightarrow P\ X = P'\ X
\end{array}$$

Moreover, there are also fundamental rules that allow for a split of defined and undefined cases and that form the bases for the generic lifter to be discussed in the next section:

$$[\![\ \text{pass } P;\ \text{pass } P';\ P\ \perp_{\mathscr{L}} = P'\ \perp_{\mathscr{L}};\ X \neq \perp_{\mathscr{L}} \Rightarrow P\ X = P'\ X\ ]\!] \Rightarrow P\ (X) = P'\ X$$

4.2 Approximating the TM_E by $LIFT_E$

Now we are ready to describe conceptually the tactic procedure. The main parts of the implementation in Isabelle/HOL are presented in the appendix, see section **??**. It is based on the set of semantic combinators SEMCOM and their theory, which has been defined elementwise in the previous sections. In order to allow a certain flexibility in the syntactic form of theorems to be lifted, we extend SEMCOM to the set CO with the set of logical connectives of our meta-language ($=$, $\wedge$, $\vee$ or $\forall$).

The $core(E)$ of a conservative theory extension E is is defined as the map

$$\{(c \mapsto c')|\textbf{constdefs}\text{ ``}c \equiv e(c')\text{''} \in axioms_of(E) \wedge constants_of(e) \subseteq CO\},$$

i.e. we filter all constant definitions that are constructed by our semantical combinators and simple logical compositions thereof.

A theorem $thm \in Th(S)$ is *liftable* iff it only contains constant symbols that are elements of $\mathrm{ran}(core(E))$ or a logical connective.

Liftable theorems can now be converted by substituting the constants in the term of *thm* along $core(E)$, i.e. we apply an inverse signature morphism constructed from $core(E)$ (note that the inverse signature morphism may not be unique; in such cases, all possibilities must be enumerated). A converted theorem may be *convertable* iff the converted term is typable in $\Sigma \uplus E$. All convertable terms thm' are fed as proof goals into a a generic tactical proof procedure that executes the following steps (exemplified with the commutativity):

1. the proof-state is initialized with thm', e.g. ((X:: α INTEGER)+ Y)=Y+X,
2. we apply extensionality and unfold the definitions for lift_1 and lift_2 yielding

   1. $\bigwedge$st. strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)

3. for each of the free variables (e.g. X and Y) we introduce a case split over definedness $DEFx$, i.e. difference of x from $\perp$ (e.g. DEF(Xst) and DEF(Y st)),

   1. $\bigwedge$st. ⟦ DEF (X st); DEF (Y st) ⟧
      $\Rightarrow$ strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)
   2. $\bigwedge$st. ⟦ DEF (X st); $\neg$DEF (Y st) ⟧
      $\Rightarrow$ strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)
   3. $\bigwedge$st. $\neg$ DEF (X st)
      $\Rightarrow$ strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)

4. we exploit the additional facts in the subgoals by simplifying with the rules for strictify$'$. This yields:

   1. $\bigwedge$st x xa. ⟦ ... ⟧ $\Rightarrow$x + xa = xa + x

5. and by applying *thm* (the commutativity on *int*) we are done.

These steps correspond to the treatment of the different layers discussed in the previous chapter: step one erases the embedding adaption layer, step two establishes case distinctions for all occurring variables and applies generic lemmas for the elimination of the semantic combinators of functional layer. In an example involving a datatype adaption layer (for example quotients like smashing in OCL), similar techniques will have to be applied.

Of course, this quite simple — since conceptual — lifting routine can be extended to a more sophisticated one that can cover a larger part of the set of convertables. For example, the combinators of the datatype adaption layer may involve reasoning over invariants that must be maintained by the underlying library functions. In our OCL theory, for example, such situations result in subproofs for

$$\llbracket\ \bot \notin \mathrm{Rep}_{Set}\mathrm{A};\ \bot \notin \mathrm{Rep}_{Set}\mathrm{B}\ \rrbracket\ \Rightarrow \bot \notin (\mathrm{Rep}_{Set}\ \mathrm{A} \cup \mathrm{Rep}_{Set}\mathrm{B})$$

Depending from the complexity of the combinators for the datatype adaption, such invariant proof can be arbitrarily complex and will require hand-proven invariance lemmas.

A particular advantage of our approach is that the lifting of theorems can be naturally extended to the lifting of the *configurations* of the automatic proof engine as well. With configuration, we mean here a number of rule sets for introduction and elimination rules for the classical reasoner `fast_tac` or `blast_tac` and sets for standard rewriting or ACI rewriting. By $LIFT_E$, these sets can be partially lifted and extended by corresponding rules on the object level. Since it is usually an expert task to provide a suitable configuration for a logic, this approach attempts to systematically extend this kind of expert knowledge from the meta-level to object level.

5 Experience gained from our OCL example

We give a short overview of the application of our approach in the typed shallow embedding of OCL into Isabelle/HOL (see [17, 18] for details). In our example scenario, we can profit a lot from the fact, that most of the functions for the datatypes Integer, Real (e.g. $=, -, /, \leq, <, \ldots$), Sequences (e.g. union, append, size, etc.), and String (e.g. concat, size, ...) can be derived in the same way as described for $+$ in the last section.

The current application of our module `thy_morpher.ML` to our OCL embedding with 85 operators produces the following statistics (based on Isabelle/HOL version 98):

```
Relevant HOL theorems : 1593
Liftable theorems     :  423
Convertable theorems  :  212
Lifted theorems       :  102
Generic theorems      :  254
```

From the 85 operators of OCL, 77 are amenable to our approach in principle. With “relevant theorems” we mean those contained in specifications imported by

the specifications containing our embedding. From our experience, improvements in the generic theorems section will lead to better results easily. In contrast, the design of new schemata of lifting proof routines is a more complex, but still rewarding task. Summing up, based on a still quite simple $LIFT_E$ technology, we successfully generated over 350 theorems which are automatically derived from the base libraries and generic theorems over semantic combinators.

6 Conclusion

We have presented a method for organizing the mass of library function definitions for typed shallow embeddings in a layered theory morphism. Moreover, we developed a technique that allows for the exploitation of this structure in a tactic-based (partial) program that lifts meta-level theorems to their object-level counterparts and meta-level prover configurations to object-level ones. Our approach can be seen as an attempt to liberate the shallow embedding technique from the "point-wise-definition-style" in favor of more global semantic transformations from one language level to another. We abstracted the underlying conceptual notions into a generic framework that shows that the overall technique is applicable in a wide range of embeddings in type systems; embedding-specific dependencies arise only from the specifications of semantic combinators (the *layers*), and technology specific dependencies from the used tactic language.

At present, the technique is limited essentially to the class of first-order Horn-clause equations; for this class, the (partial) program succeeds in our application in all cases in our non-trivial application language. Although a more precise characterization of success is impossible here due to the generality of the framework, we believe that the approach will be applicable for language embeddings for SML, Haskell or Z [16] with similar success since the underlying semantic combinators are the same. Additionally, our implementation of $LIFT_E$ will also be reusable. The same holds for many basic generic theorems over semantical combinators from the embedding adaption layer, the functional adaption layer and — to a lesser extent — the data adaption layer. In principle, the overall construction is also applicable for other higher-order typed theorem proving systems such as Coq [8] or ALF [33]; however, the theories over the semantic combinators and the core of the tactic procedure will have to be adapted to these frameworks.

Besides the obvious need for more generic theorems and more powerful lifting proof procedures, in particular for formulae like $\neg\forall x : A.Px = \exists x : A.\neg Px$, the potential of our approach for untyped shallow or even deep embeddings should be explored. This means, that similarly to invariance proofs of data adaption operators, automatic proofs for the maintenance of well-typing have to be constructed, whereas in a deep embedding, the invariance of binding correctness ("no name-clashes") has also be handled in these proof routines. Beyond the obvious increase of complexity, it seems unclear what kind of limitations for such a setting will arise.

References

[1] Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)

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

[3] Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. Prentice Hall International Series in Computer Science (1992)

[4] Kolyang, Santen, T., Wolff, B.: A structure preserving encoding of Z in Isabelle/HOL. In von Wright, J., Grundy, J., Harrison, J., eds.: TPHOLs. LNCS 1125, Springer (1996)

[5] Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge Press (1993)

[6] Nipkow, T., von Oheimb, D., Pusch, C.: μJava: Embedding a programming language in a theorem prover. In Bauer, F.L., Steinbrüggen, R., eds.: Foundations of Secure Computation. Volume 175 of NATO Science Series F: Computer and Systems Sciences., IOS Press (2000) 117–144

[7] `http://www.nuprl.org`.

[8] `http://pauillac.inria.fr/coq/`.

[9] `http://isabelle.in.tum.de`.

[10] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)

[11] `http://www.ora.on.ca/z-eves/welcome.html`.

[12] `http://svrc.it.uq.edu.au/pages/Ergo.html`.

[13] `http://i11www.ira.uka.de/~kiv/`.

[14] Reetz, R.: Deep Embedding VHDL. In E.T. Schubert, P.J. Windley, J. Alves-Foss, eds.: 8th International Workshop on Higher Order Logic Theorem Proving and its Applications. Volume 971 of Lecture Notes in Computer Science., Springer (1995) 277–292

[15] Ozols, M.A., Eastaughffe, K.A., Cant, A., Collignon, S.: DOVE: A tool for design modelling and verification in safety critical systems. In: 16th International System Safety Conference. (1998)

[16] Brucker, A.D., Rittinger, F., Wolff, B.: HOL-Z 2.0: A proof environment for Z-specifications. Journal of Universal Computer Science **9** (2003)

[17] Brucker, A.D., Wolff, B.: A proposal for a formal OCL semantics in Isabelle/HOL. In Muñoz, C., Tahar, S., Carreño, V., eds.: Theorem Proving in Higher Order Logics. Number 2410 in LNCS. Springer (2002) 99–114

[18] Brucker, A.D., Wolff, B.: HOL-OCL: Experiences, consequences and design choices. In Jezequel, J.M., Hussmann, H., Cook, S., eds.: UML 2002: Model Engineering, Concepts and Tools. Number 2460 in LNCS. Springer (2002)

[19] OMG: Object Constraint Language Specification. [22] chapter 6

[20] Warmer, J., Kleppe, A.: The Object Contraint Language: Precise Modelling with UML. Addison-Wesley (1999)

[21] Warmer, J., Kleppe, A., Clark, T., Ivner, A., Högström, J., Gogolla, M., Richters, M., Hussmann, H., Zschaler, S., Johnston, S., Frankel, D.S., Bock, C.: Response to the UML 2.0 OCL RfP. Technical report (2001)

[22] OMG: Unified Modeling Language Specification (Version 1.4). (2001)

[23] Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. Journal of the ACM (JACM) **39**(1) (1992) 95–146

[24] Nipkow, T.: Order-sorted polymorphism in Isabelle. In Huet, G., Plotkin, G., eds.: Logical Environments. (1993) 164–188

[25] Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5** (1940) 56–68
[26] Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Academic Press (1986)
[27] Barendregt, H.: Lambda Calculi with Types. In: Handbook of Logic in Computer Science. Clarendon Press (1992) 117–309
[28] Frank Pfenning, C.E.: Higher-order abstract syntax. In: PLDI 1988. (1988) 199–208
[29] G. Huet, B.L.: Proving and applying program transformations expressed with second order patterns. (Acta Informatica)
[30] Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In Stavridou, V., Melham, T.F., Boute, R.T., eds.: Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience. Volume A-10 of IFIP Transactions., Nijmegen, The Netherlands, North-Holland/Elsevier (1992) 129–156
[31] Wadler, P.: Comprehending monads. In: Proc. 1990 ACM Conference on Lisp and Functional Programming. (1990)
[32] King, D.J., Wadler, P.: Combining monads. In: Glasgow functional programming workshop. (1992)
[33] Altenkirch, T., Gaspes, V., Nordström, B., von Sydow, B.: A User's Guide to ALF. Chalmers University of Technology, Sweden. (1994)