ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lukas Brügger

Proof Support for IMP++ in HOL-OCL

A Typed Verification Approach for Object-oriented Programs

Master Thesis
ETH Zürich
July 29, 2007

Supervisors: Achim Brucker, Burkhart Wolff
Professor: David Basin

Abstract

We present a typed approach for verifying object-oriented programs. The approach is based on a Hoare logic and is fully integrated into HOL-OCL, a proof environment for object-oriented specifications. With this integrated method, we can bridge the gap between the big-step semantics of specification languages like UML/OCL and a small-step semantics of a program verification method. As HOL-OCL is based on a conservative embedding of UML/OCL into higher-order logic and we only extend it conservatively, we can ensure the consistency and type-safety of our calculus.

Zusammenfassung

In dieser Arbeit präsentieren wir einen getypten Ansatz zur Verifikation objektorientierter Programme. Der Ansatz basiert auf einer Hoare Logik und ist vollständig in HOL-OCL integriert, eine Beweisumgebung für objektorientierte Spezifikationen. Mit dieser Integration verbinden wir die big-step Semantik einer Spezifikationssprache wie UML/OCL mit einer small-step Semantik einer Programmverifikationsmethode. Da HOL-OCL auf einer konservativen Einbettung von UML/OCL in HOL basiert und wir dieses Rahmenwerk nur konservativ erweitern, können wir die logische Konsistenz und Typsicherheit unseres Kalküls garantieren.

Contents

1 Introduction

Computer systems are becoming both more complex and increasingly important in our daily lives. Therefore it is of big importance to construct them in a correct and reliable way. During the last decades, many methods have been proposed to improve the software engineering process and to improve the assurance level of source code. Most of these methods employ some sort of formal methods, as it is today widely acknowledged that "a more mathematical approach is needed for software to progress much" [9].

Formal verification of a program is the process of proving that a software does exactly what is stated in the program specification. Programming language semantics like Hoare logic, which we use in this thesis, are concerned with the meaning of programming language constructs. They reason about the system state after every single command, which is why we call them a small-step semantics. Object-oriented specification languages like UML/OCL on the other hand, use a big-step semantics, as they only describe the overall result of an operation. In this thesis, we will formally bridge the gap between the big-step semantics of a specification and the small-step semantics of its implementation.

Verification systems can either be defined conservatively or axiomatically. Our system will be defined completely conservatively, i.e. HOL will only be extended with constant definitions and proved lemmas. Using a conservative approach, the consistency and correctness can be assured directly. There are many verification methods which are defined conservatively. However, most of these methods have problems with object-orientation, today's most widely used programming paradigm. Many of the most prominent features of object-orientation like classes, inheritance, sub-typing, objects and references are deeply intertwined and complex concepts that are quite remote from the platonic world of traditional logic. There is a tangible conceptual gap between the verification of functional and imperative programs on one hand and object-oriented programs on the other. Most existing verification systems for object-oriented programs are defined axiomatically. In this thesis, we present an object-oriented verification method which is defined entirely conservatively.

Our method is fully integrated into the specification environment HOL-OCL. This framework enables reasoning over object-oriented data structures supporting sub-typing and single inheritance. HOL-OCL is based on a shallow embedding of UML/OCL data models in higher-order logic. This embedding approach leads to a methodology which is completely type safe. Our programming language IMP++ employs the same model of an object-oriented store as the one provided by the UML model. The big-step semantics are provided through an OCL specification. IMP++ provides the implementation of the operations and the small-step semantics through a Hoare calculus. This enables us to prove that the implementation conforms to the specification. With our integrated approach, we are able to bridge the gap between specification and code verification and widen the applicability of the model transformation approach.

Our method is based on a so called shallow embedding, where object-language binding and typing are represented directly in the typing machinery of the meta-language. This is in contrast to many approaches which are based on a deep embedding and therefore have to deal with a heavy syntactic bias in form of side-conditions over binding and typing issues.

The thesis is structured as follows. In Chapter 2 we introduce the framework on which our work will be based and present the necessary background of the system and of the semantics of programming languages. In Chapter 3 we present our programming language IMP++ together with its denotational semantics and the Hoare calculus. We then show how HOL-OCL has to be extended to support program verification with the help of a running example in Chapter 4. In Chapter 5 we present the calculus needed to reason about an implementation and present a proof of an invariant of a program. Finally, we draw the conclusion, present related work and give suggestions for future work.

2 Background

We present our verification approach using a running example throughout the thesis. We model a linked list. A class called `Node` has an attribute `next` which is a pointer to the next element of the list. List operations like insert and delete could then be defined on this class. The class has also the attribute `size` of type Integer. The nodes could then store an arbitrary content with defining a respective subclass of `Node`. In our example, we add the class `Cnode`, which has an attribute called `color` of type Boolean.

We would like to have an invariant in our model which states that two neighbouring `Cnodes` in the list must never have the same value in their `color` attribute.

The following program, given here in pseudo-code, constructs a cyclic list:

```
N1 := New(Cnode);
N2 := New(Cnode);
N1 := N1.set_color true;
N2 := N2.set_color false;
N1 := N1.set_next N2;
N2 := N2.set_next N1;
return := N1;
```

The list which gets created with this program can be seen in Figure 2.1. This Figure informally shows that the program indeed fulfils the invariant. Later we will prove this formally.

In the following we provide the necessary background for the thesis.

2.1 UML/OCL

The Unified Modelling Language (UML) [8] is a widely used graphical specification language for object-oriented systems. It provides a variety of diagram types for describing

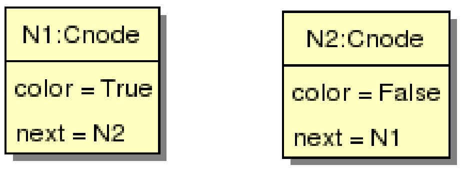

Figure 2.1: The linked list which gets created through the program

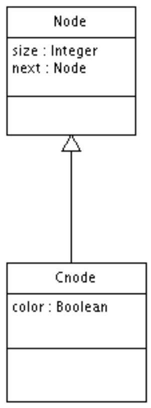

Figure 2.2: A class diagram with two classes

system properties. The most important diagram type is the class diagram for modelling the data model of a system. The class diagram of our example is shown in Figure 2.2. We can see the two classes with their attributes.

The UML as a graphical language is not sufficient to specify the invariant. Therefore, UML diagrams are heavily accompanied by constraints written in the Object Constraint Language OCL [7]. With OCL, we have the possibility to add preconditions, postconditions and invariants to the elements of a UML model.

Our invariant would be formalised as follows in OCL:

```
context Cnode
inv flip:  self.color <> self.next.color
```

For all objects `self` of type `Cnode`, we specify an invariant which we call `flip`, which says that for each `self`, its `color` attribute must not have the same value as the `color` attribute of its `next` object.

Specification languages like UML/OCL describe what is often called a big-step semantics. They describe the overall result of the execution. Code verification later will use a small-step semantics, reasoning about every single step of an execution, where usually an invariant can not be established through one single step. These two views then have to be integrated later.

A distinguishing feature of OCL is that whenever an OCL expression is being evaluated, there is a possibility that one or more of the subexpressions in the expression are undefined. In that case, the complete expression will usually be undefined. Conversely, this also means that if an expression is defined, each of its subexpressions must be defined as well. The only exceptions to this rule are the Boolean operators, and actually OCL is a strong three-valued Kleene logic. This explicit dealing with undefinedness is a distinguishing feature of OCL, which separates it from most other specification languages.

2.2 Isabelle/HOL

Isabelle [6] is a generic, LCF-style theorem prover implemented in SML. It allows to build SML programs performing symbolic computations over formulae in a logically safe way. Isabelle/HOL supports conservativity-checks of definitions, datatypes, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableau provers.

Higher-order logic (HOL) [1] is a classical logic with equality enriched by total polymorphic higher-order functions. The type constructor for the function space is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ are also written as $[\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow$ bool, where bool is the fundamental logical type.

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

In the following, we give a short overview of the Isabelle syntax we use in this thesis. Further information can be found in [6].

New constant definitions can be introduced using **constdefs**. For example the following definition adds the xor operator with the type bool $\Rightarrow$ bool $\Rightarrow$ bool:

constdefs
  xor :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
  xor A B $\equiv A \wedge \neg B \vee \neg A \wedge B$

In traditional mathematics, a theorem, or lemma, is usually written in the following form:

$$\frac{A \longrightarrow B; A}{B}$$

In Isabelle, this is written as:

lemma *example*: $[\![A \longrightarrow B;\ A]\!] \Longrightarrow B$

Lemmas are then proved with a sequence of rule applications, usually in backward style. These rule applications are written as:

apply (*rule other_example*)

Where *other_example* is another lemma. Once a lemma is proved, it can be used as a rule in other proofs.

Using *rule_tac* we can give explicit substitutions of the rule to be applied.

Isabelle is able to conduct large parts of a proof automatically. Two commands which

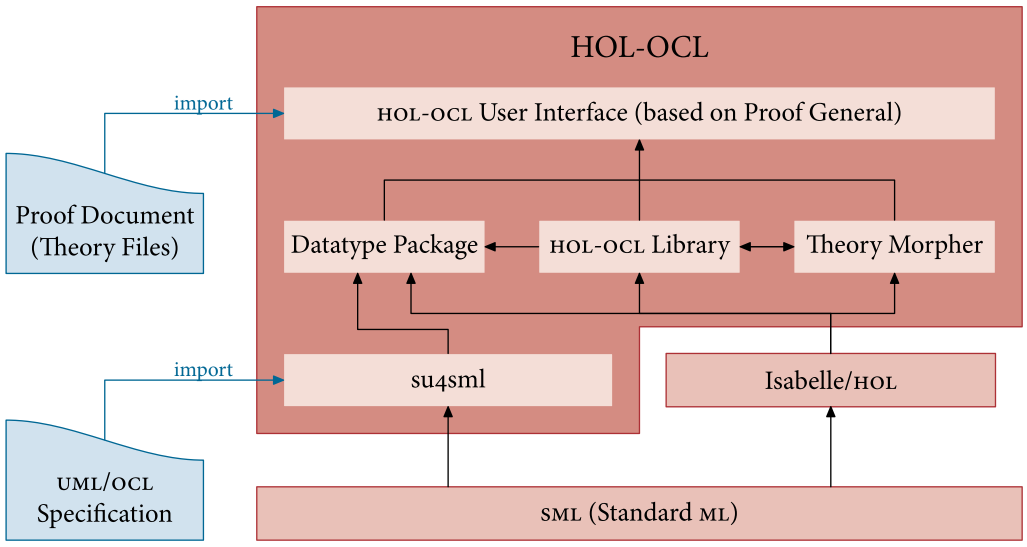

Figure 2.3: Overview of the HOL-OCL Architecture

are used to instruct Isabelle to use automatic tactics are *auto* and *simp*. The latter is basically a rewriter. It can be extended by adding additional rules to the current simplifier set.

Isabelle also provides a powerful tool for document generation directly from Isabelle theories. This method was used for this document as well. This ensures that all the definitions we provide are type correct and all the lemmas we present and use are actually proved inside the theories, even if it the proof script has been suppressed from the output.

2.3 HOL-OCL

HOL-OCL [4] is an interactive proof environment for UML/OCL. It's mission is to give the term "object-oriented specification" a formal semantic foundation and to provide effective means to formally reason over object-oriented models. It is implemented as a conservative, shallow embedding of OCL into Isabelle/HOL. HOL-OCL allows one to reason over OCL specifications, to refine OCL specifications, and builds the basis for further tool support, e.g. for the automatic test-case generation or for program verification.

We refrain here from giving a complete description of HOL-OCL but rather give a short overview over the parts used in this thesis.

The architecture of HOL-OCL is shown in Figure 2.3. Specifications written in UML/OCL are imported into the model repository su4sml, written in SML. The theory morpher for lifting proven lemmas from the HOL to the OCL level is developped on top of Isabelle/HOL. The datatype package encodes the UML/OCL models and proves already several properties over the specification. The formal analysis of the specification is then

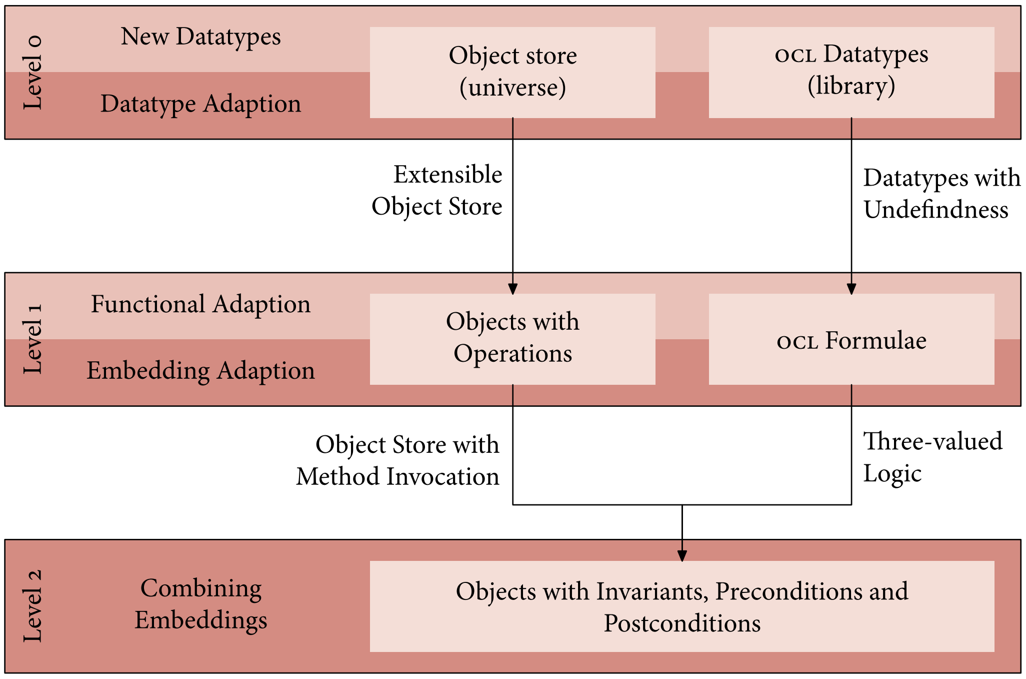

Figure 2.4: Structure of the embedding

carried out using a user interface based on Proof General. In the context of this thesis, we will extend the Datatype Package and extend the HOL-OCL Library with support for program verification, the other parts of the system will remain untouched.

The shallow embedding of UML/OCL in HOL-OCL is organized into levels. Figure 2.4 gives an overview of this modular architecture.

Level 0: This level defines the ground work for the embedding. It defines both an extensible object store and the datatypes for OCL. After this level, there is a rich library of datatypes supporting undefinedness.

Level 1: This level adapts the functional behavior and finalizes the embeddings. It provides an embedding of an extensible object store with operation invocation and a three-valued logic.

Level 2: This level combines the two embeddings, i.e. it introduces the context of the constraints and defines the semantics of objects and method invocations with respect to the validity of the corresponding preconditions, postconditions and invariants.

With the datatype adaption on level 0, the gap between the types of UML/OCL and HOL has to be bridged. For example, to handle the undefinedness inherent in OCL, HOL-OCL defines a type class $\alpha ::$ bot for all types α that provide an exceptional element $\perp$; for each type in this class, a test for definedness is available via DEF $x \equiv (x \neq \perp)$. The HOL type constructor τ *up* assigns to each type τ a type *lifted* by $\perp$. Thus, each type

α *up* is member of the class bot. The function $\llcorner\_\lrcorner : \alpha \Rightarrow \alpha$ *up* denotes the injection, the function $\ulcorner\_\urcorner : \alpha$ *up* $\Rightarrow \alpha$ its inverse for defined values. Based on these definitions, partial functions, i.e. $\alpha \rightharpoonup \beta$, can be represented as total functions of type $\alpha \Rightarrow \beta$ *up*.

The encoding of objects and the object store is very involved, details can be found in [4]. First, for each model a universe is created which will contain all the objects. As a distinguishing feature of HOL-OCL, this universe is constructed in a way which supports incremental proofs. If we extend an existing class model with new classes, we don't have to replay the proof scripts. Details of how this works can be found in [2].

For each class in our model, a class type is created. An object of a specific type is basically represented by a tuple. For example, the type of `Cnode` in our example is:

```
('α) Cnode =
    ((OclAny_key × oid) ×  ((Node_key × int up ×  oid) ×
    ((Cnode_key × bool up) ×  'α up) up) up) up
```

Where OclAny_key, Node_key and Cnode_key are "tag types" which are necessary to build a stongly typed universe. The oids are the object identifiers. We assume that each object has a unique oid. The first one appearing in the tuple above is the oid of the object itself, the second one the oid of the `next` attribute. The level 0 representation is therefore not type-safe. The `next` attribute could denote any object. An object can either be completely undefined, or only parts of it. This is the reason for all the liftings in the type. An object has a non-smashed semantics: it can be defined while some of its parts are undefined. This is why the lifted types are used for the attributes.

Additionally to the user-defined classes, there is always the class `OclAny`, which is the superclass of all objects.

Finally, the type variables denote the possible extensions of a class. Here, $'\alpha$ allows for extension with new classes by inheriting from `Cnode`. This support for extensions is a distinguishing feature of HOL-OCL. It means that existing proofs do not have to be rebuild if we add a new class to an existing system - a necessary condition if the tool is supposed to be used in a realistic object-oriented setting.

So far, HOL-OCL does not have any programming language semantics. Therefore, it can not be used for program verification. With our thesis, we extend HOL-OCL with a programming language semantics.

2.4 Hoare Logic

If we want to provide a method for verifying source code, we have to deal with the formal semantics of programming languages. The approach we are using in this thesis is a Hoare logic.

Hoare logic is a formal system to reason about the correctness of computer programs. It has been introduced by C.A.R Hoare almost forty years ago.

The idea of Hoare logic is to relate "legal states" before and after program execution. A set of legal states is an assertion. The key construct is then a Hoare triple of the form

$$\models \{P\}c\{Q\}$$

where `P` and `Q` are assertions and `c` a command. The intuitive meaning of such a triple is "if `P` holds for some state `s` and `c` terminates and reaches state `t` then `Q` must hold for `t`". Such a triple is also called a *partial correctness assertion* because it does not make a statement about what happens if command `c` fails to terminate. A Hoare calculus then provides rules for each construct of the programming language. These rules are called *Hoare rules.*

There exists a formalisation of Hoare logic for an imperative language called IMP in Isabelle [5], which follows the lines of an important textbook on the topic [12]. We provide an extension of IMP, called IMP++, with support for object-orientation.

However, we see the Hoare logic only as an intermediate step. We use it as a tool to show the feasibility of our approach. It would be too cumbersome to proof a non-small program with this methodology. However, it provides the basis for a verification condition generator which can be implemented in the future.

As noted earlier, in a big step semantics it is sufficient to check the invariant before entering and after leaving the method. A small step semantics like the Hoare logic has to observe each single command. We can't establish the invariant through one single step during program execution, but rather have to prove it holds when leaving the method.

2.5 Workflow

Having presented the necessary background, we can now outline the overall workflow of the system:

- The model, i.e. object-oriented datatypes and their operations, is specified using UML and OCL
- The model is imported into HOL-OCL. The datatype package creates the definitions and proves some lemmas
- HOL-OCL provides the means to reason about the specification
- Using IMP++ we can implement the specified program
- HOL-OCL and IMP++ use the same object store
- The HOL-OCL library is extended with theorems useful for program verification
- We can reason over the program using Hoare logic, e.g. show that the invariant is fulfilled at the end of the method
- Finally, we can establish that our program conforms to the specification

3 IMP++

In this Chapter we present our object-oriented programming language, IMP++, and show how the rules for denotational and axiomatic semantics can be derived. IMP++ is defined as an extension of IMP, an imperative programming language for which a semantics and a Hoare calculus were developed inside Isabelle-HOL [5]. Basically, IMP++ can be seen as IMP enriched by object-orientation. Big parts of this formalisation have already existed in HOL-OCL. For the context of this thesis, we extended it in some parts and made minor changes.

3.1 The Language

As IMP, IMP++ is a deep embedding and the syntax is introduced with a datatype definition. As a prerequisite we need a type for Boolean expressions and commands.

```
types
  'α bexp  = 'α state ⇒ bool up
  'α cmd   = 'α state ⇒ 'α state up
```

We can then introduce the syntax as a datatype definition and directly provide syntax translations to make the language more readable.

```
datatype
  'α com =
        SKIP
      | Cmd    'α cmd
      | Semi   'α com 'α com           (_; _  [51, 50] 50)
      | Cond   'α bexp 'α com 'α com  (IF _  THEN _  ELSE _   60)
      | While  'α bexp 'α com          (WHILE _  DO _   60)
```

The language consists of only five different constructs:

- *SKIP* denotes the empty, successfully terminating, command.
- *Cmd* is a generic command that takes as argument a function *'α Cmd*, which is a function from a state to a state lifted by $\perp$. Thus *Cmd* is allowed to return the $\perp$ state, which is used for modelling exceptions.
- *Semi* is the sequential composition. It is written as _ ; _.
- *Conditional* is the usual if-then-else construct. It is written as *IF _ THEN _ ELSE _*.

- *While* is the traditional while loop. It is written as *WHILE _ DO _*.

The language seems very tiny and not very object-oriented at a first glance. But actually almost all constructs we need for an object-oriented programming language can be integrated into the generic *Cmd* slot, notably all setters and getters. The object-orientation of the language mainly comes from the model of the state we are using. This state will be presented in Section 4.1. The *Cmd* slot allows arbitrary modifications of the object-oriented state, which allows to keep the language as simple as possible while supporting a large subset of a real object-oriented language.

Another difference to the imperative IMP is the support for undefined states. The *Cmd* slot is defined as returning a state lifted by the $\perp$ element. This can be used to model exceptions, an important object-oriented concept. To emphasize that this is used to model exceptions, we introduce a constant for denoting the erroneous state.

constdefs
  err :: $'\alpha$ *state up* $\Rightarrow$ *bool up*
  err $\equiv$ $(\lambda\ x.\ \llcorner x{=}\perp\lrcorner)$

Currently, we only have one type of exceptions. The lemma *err2sem*: "$(\sigma \vDash err) = (\neg\ DEF\ \sigma)$" states that saying *err* is true in some state σ is equivalent to saying that σ is undefined. This allows reusing the calculus for undefinedness of the HOL-OCL library. An exception could for example occur if we try to update an object that does not exist. In Chapter 4 we will see an example of an update which returns an erroneous state.

3.2 Denotational Semantics

Denotational semantics provide the basis for Hoare logic. The idea is to explain recursion as fixpoint construction on the semantic domain. The semantic domain is a state relation. Contrary to IMP, we use lifted states and the type of our relation is as follows:

types $'\alpha$ *com_den* = ($'\alpha$ *state up* $\times$ $'\alpha$ *state up*) *set*

As a consequence of our decision to extend the traditional relation of denotational semantics, we need to add an extension of the traditional relation composition. Despite the checks for definedness, this definition is as usual. The relation is defined such that the $\perp$ element will be propagated.

constdefs
  O_up_comp :: ($'\beta$ *up* $\times$ $'\gamma$ *up*) *set* $\Rightarrow$ ($'\alpha$ *up* $\times$ $'\beta$ *up*) *set* => ($'\alpha$ *up* $\times$ $'\gamma$ *up*) *set*
      (**infixl** *O'_up 55*)
  r O_up s $\equiv$ $\{(\perp,\perp)\}\ \cup$
      $\{(x,\ z).\ DEF\ x \wedge (\exists\, y.\ DEF\ y \wedge (x,\ y) \in s \wedge (y,\ z) \in r)\}\ \cup$
      $\{(x,\ z).\ DEF\ x \wedge (\exists\, y.\ \neg DEF\ y \wedge (x,\ y) \in s \wedge z{=}\perp)\}$

We now need to specify which states are legal. We do this as usual with the help of the semantic function *C*, which is a primitive recursion over the syntax. This definition is fairly standard, but enriched by the cases for undefined states. Gamma is also the usual but extended approximation functional for the least fix-point operator *lfp*.

constdefs
$Gamma :: [{}'\alpha\ bexp,\ {}'\alpha\ com\_den] => ({}'\alpha\ com\_den => {}'\alpha\ com\_den)$
$Gamma\ b\ cd \equiv (\lambda phi.\ \{(s,t).\ (s{=}\bot \vee\ b \ulcorner s \urcorner = \bot)\ \wedge\ t{=}\bot\} \cup$
$\quad \{(s,t).\ DEF\ s \wedge DEF(b \ulcorner s \urcorner) \wedge b \ulcorner s \urcorner = \llcorner True \lrcorner\ \wedge (s,t) \in (phi\ O\_up\ cd)\} \cup$
$\quad \{(s,t).\ DEF\ s \wedge DEF(b \ulcorner s \urcorner) \wedge b \ulcorner s \urcorner = \llcorner False \lrcorner \wedge s{=}t\})$

consts
$C :: {}'\alpha\ com => {}'\alpha\ com\_den$

primrec
C_skip: $C\ SKIP = Id$
C_cmd: $C\ (Cmd\ f) = \{(s,t).\ s{=}\bot\ \wedge t{=}\bot\} \cup$
$\quad \{(s,t).\ DEF\ s \wedge t{=}\ f \ulcorner s \urcorner\}$
C_comp: $C\ (c0\ ;\ c1) = C(c1)\ O\_up\ C(c0)$
C_if: $C\ (IF\ b\ THEN\ c1\ ELSE\ c2) =$
$\quad \{(s,t).\ (s{=}\bot \vee\ b \ulcorner s \urcorner = \bot)\ \wedge\ t{=}\bot\} \cup$
$\quad \{(s,t).\ DEF\ s \wedge DEF(b \ulcorner s \urcorner) \wedge b \ulcorner s \urcorner = \llcorner True \lrcorner \wedge (s,t) \in C\ c1\} \cup$
$\quad \{(s,t).\ DEF\ s \wedge DEF(b \ulcorner s \urcorner) \wedge b \ulcorner s \urcorner = \llcorner False \lrcorner \wedge (s,t) \in C\ c2\}$
C_while: $C(WHILE\ b\ DO\ c) = lfp\ (Gamma\ b\ (C\ c))$

Like the relation composition, this definition is modelled in a way such that the $\bot$ state gets propagated. So once we are in an error state, we can not reach a defined state anymore. This fact is proved by the following lemma. If there is a command c which relates the states $(\bot,\sigma)$, σ has to be $\bot$.

lemma *bottom_prop*:
$(\bot,\sigma) \in C(c) = (\sigma = \bot)$

A standard example for denotational semantics is a proof of the equality of the following two program fragments. Such equality results justify program transformations. The proof is quite short. It is important to note that this fact can be derived from above definitions. It is not added as an axiom to our calculus like in some traditional approaches, but proved as a theorem.

lemma *C_While_If*:
$C(WHILE\ b\ DO\ c) = C(IF\ b\ THEN\ c\ ;\ WHILE\ b\ DO\ c\ ELSE\ SKIP)$

3.3 Hoare Rules

The idea of a Hoare logic is to relate legal states before and after program execution. We can now state the Hoare rules for IMP++, which show which states can be related through which command. We have to start with the type definition of the assertions. As in [5], we take a semantic view of assertions. This way the logic used for assertions is the same as the one of the underlying proof system and we don't have to invent a new one and worry about expressiveness.

An assertion is modelled as a predicate on the state, which is lifted in our case.

types *'α assn = 'α state up ⇒ bool*

The definition of a Hoare triple is then straightforward:

constdefs
  hoare_valid :: [*'α assn*, *'α com*, *'α assn*] => *bool*
                                                  (⊨ {(*1*_)}/ (_)/ {(*1*_)} *50*)
  $\models \{P\}c\{Q\} \equiv \forall\ s\ t.\ (s,t) \in C(c) \longrightarrow P\ s \longrightarrow Q\ t$

Again, this definition is fairly standard. It is important to note that we deal with partial correctness only.

Unlike [5], where the Hoare rules are introduced using an inductive definition, we prove the rules directly as lemmas. The detour via a derivability notion is unnecessary as we focus on correctness proofs and not completeness.

We start with giving the rules for each command of our language. Again, these lemmas are proved, they are not introduced as axioms.
The rule *skip_hoare* for the *SKIP* construct states that if an assertion is true of the state before the execution of *SKIP*, it will be true afterwards. This has to hold as *SKIP* is modelled to be the empty command, which in particular will not change the state.

$$\models \{P\}\ \mathit{SKIP}\ \{P\}$$

The rule *semi_hoare* is standard: if $\{P\}c\{Q\}$ and $\{Q\}d\{R\}$ are valid, so must be $\{P\}c;d\{R\}$.

$$\frac{\begin{array}{c}\models \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge P\ \sigma\}\ c\ \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge Q\ \sigma\} \\ \models \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge Q\ \sigma\}\ d\ \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge R\ \sigma\}\end{array}}{\models \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge P\ \sigma\}\ c;\ d\ \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge R\ \sigma\}}$$

The rule *cmd_hoare* for the Cmd construct is similar to the usual rule for assignment. In the assertion before the command, we have to add the condition that *f* is defined in that state. This is written as $\ulcorner\sigma\urcorner \vDash \partial f$.

$$\models \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge \ulcorner\sigma\urcorner \vDash \partial\ f \wedge\ Q\ (f\ \ulcorner\sigma\urcorner)\}\ \mathit{CMD}\ f\ \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge\ Q\ \sigma\}$$

The *if_hoare* rule has to be extended with the cases where the expression is undefined. Otherwise it is standard, it represents the case split.

$$\frac{\begin{array}{c}\models \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge P\ \sigma \wedge \ulcorner\sigma\urcorner \vDash b \wedge \ulcorner\sigma\urcorner \vDash \partial\ b\}\ c\ \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge\ Q\ \sigma\} \\ \models \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge P\ \sigma \wedge \ulcorner\sigma\urcorner \vDash \neg\ b \wedge \ulcorner\sigma\urcorner \vDash \partial\ b\}\ d\ \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge\ Q\ \sigma\}\end{array}}{\models \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge \ulcorner\sigma\urcorner \vDash \partial\ b \wedge P\ \sigma\}\ \mathit{IF}\ b\ \mathit{THEN}\ c\ \mathit{ELSE}\ d\ \{\lambda\sigma.\ \neg\ \sigma \vDash \mathit{err} \wedge\ Q\ \sigma\}}$$

The next rule, *exn_hoare*, is a consequence of our choice that the error state gets always propagated. With no sequence of commands we can reach a defined state from an undefined one.

$$\models \{\lambda\sigma.\ \sigma \vDash \mathit{err}\}\ c\ \{\lambda\sigma.\ \sigma \vDash \mathit{err}\}$$

The rule *conseq_hoare* is also standard in a Hoare calculus. We can always strengthen the precondition or weaken the postcondition.

$$\frac{\forall s.\ P'\ s \longrightarrow P\ s \qquad \models \{P\}\ c\ \{Q\} \qquad \forall s.\ Q\ s \longrightarrow Q'\ s}{\models \{P'\}\ c\ \{Q'\}}$$

4 Extending HOL-OCL

In this Chapter we show how the HOL-OCL framework has to be extended to support program verification. Only two parts of the system as shown in Figure 2.3 have to be extended: the datatype package and the HOL-OCL library.

We will continue to use the level structure of the system as outlined in Section 2.3. Figure 4.1 shows how our extensions fit into this level hierarchy. The level 0, its datatypes and object store will remain the same. We just have to extend it by several definitions and lemmas which were not there yet. It is actually one of the strength of the object-store of HOL-OCL that it can also handle the encoding of objects of a real object-oriented programming language. Level 1, which adds the functional behaviour, has to be adapted. It is not the same for IMP++ and HOL-OCL. Properties proved on this level in IMP++ will however have a well defined connection to the level 1 of HOL-OCL. This interfacing will take place on level 2 where the support for preconditions, postconditions and invariants is added.

Actually, there is an intermediate step between level 0 and level 1. The real level 0 only speaks of datatypes and objects, while on level 1 these objects live in a state. We therefore have the concepts of level 0, followed by level 0 with state, and then level 1.

In this Chapter, we will explain the details of the implementation following the level structure. First we introduce the state, then show the extensions of level 0, followed by level 1 and level 2. All of these will be split into a general part and a part where we explain the implementation for the example we introduced in Chapter 2. It is envisaged that most of the following definitions and lemmas will be generated by the encoder while importing the model.

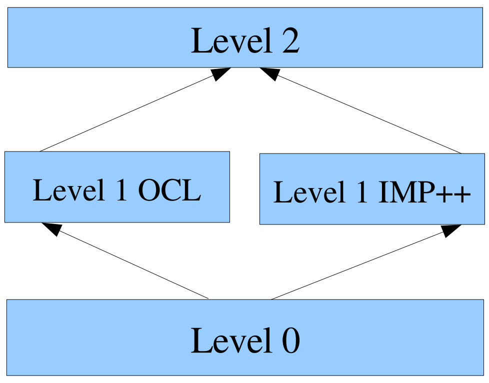

Figure 4.1: Structure of the embedding

4.1 State

Definition

We've already seen that states are a crucial part of our programming language. Almost all the commands of the language are just transitions on states, and the state is what essentially makes the language object-oriented. Intuitively, a state is a store of all existing objects. The way one single object is stored will be outlined in the next Section. We model the state as being the store of all the objects which currently exist in our program.

HOL-OCL creates a referential universe where each object has a unique object identifier, the oid. We have seen in Section 4.1, that this oid is stored in the object itself. The idea of object identifiers is that they unambiguously denote an object. Mainly, two different objects must never have the same oid. This allows to use the oids as references to objects and paves the way for modelling the state as a mapping from the set of oids to the universe of objects.

The type oid is in the type class *bot* and can thus also be undefined. This concept is similar to the `null` reference in Java, where it is a special value that is a reference to nothing, or an absence of a reference. To make this concept more explicit, we add a constant which denotes the undefined oid.

constdefs
  $NULL :: oid$
  $NULL \equiv \bot$

As the oid is always stored at the same place of an object tuple, the operator which returns the oid of an object can be defined once and for all for all object types.

constdefs
  $OclOidOf0 :: {'\alpha}\ OclAny\_0 \Rightarrow oid$
  $OclOidOf0\ X \equiv snd(fst \ulcorner X \urcorner)$

Although the state is basically a mapping from oids to the universe, this definition is too broad for the semantics we want to provide. We need a couple of additional requirements. Especially, it is a very important concept of our approach that we have a typed state, where each object in the state must be of correct type.

There are three basic requirements we have to impose on the states:

1. The oid *NULL* must not be a valid reference; as a consequence, dereferentiation of *NULL* will always result in an undefined value.

2. All elements in the range of the state must be defined, be it objects or values.

3. Objects referred from an oid must contain this *oid* in their second component. Thus, there is a "one-to-one"-correspondence between objects and their *oid*.

The last component is a key-requisite of the referential universe construction.

These three constraints can be formalized as follows.

constdefs
  $correct\_state :: (oid \rightharpoonup ('\alpha\ U)) \Rightarrow bool$
  $correct\_state\ \sigma \equiv$
   $(NULL \notin dom\ \sigma\ \wedge$
   $(\forall\ obj \in ran\ \sigma.(level0.oclLib.is\_OclAny\_univ\ obj \longrightarrow DEF(level0.oclLib.get\_OclAny\ obj))\ \wedge$
   $(is\_Real\ obj \longrightarrow DEF(get\_Real\ obj))\ \wedge$
   $(is\_Integer\ obj \longrightarrow DEF(get\_Integer\ obj))\ \wedge$
   $(is\_Boolean\ obj \longrightarrow DEF(get\_Boolean\ obj))\ \wedge$
   $(is\_String\ obj \longrightarrow DEF(get\_String\ obj)))\ \wedge$
   $(\forall\ oid \in dom\ \sigma.\ level0.oclLib.is\_OclAny\_univ\ (the\ (\sigma\ oid)) \longrightarrow$
   $OclOidOf0(level0.oclLib.get\_OclAny\ (the\ (\sigma\ oid))) = oid))$

In the part which assures the definedness of all the objects in the state, we add the cases for objects of type Real, Integer, Boolean, and String, as they are treated specially in HOL-OCL.

This definition forms the so called state-invariant. It defines the set of valid states.

Using this state-invariant, we can define the states using a type definition. Using a type definition, an existing type can be turned into a new type. The new type is specified to be isomorphic to some non-empty subset of an existing type. It then has to be proved that this new type is in fact not empty, usually this is done by providing a witness.

typedef $'\alpha\ state = Collect\ (correct\_state: : (oid \rightharpoonup ('\alpha\ U)) \Rightarrow bool)$
  by$(rule\_tac\ x{=}\lambda\ x.\ None$ **in** $exI,\ simp\ add{:}\ correct\_state\_def\ OclOidOf0\_def)$

We see that states are defined as the subset of those maps from oid to universe which satisfy the invariant. This view of the state as a constrained object store over object-oriented models is of crucial importance to IMP++, as the object-orientation of the language basically comes from the state.

Using a type definition, Isabelle automatically proves the most important lemmas for the projection and injection. As we wouldn't want to switch all the time between the two representations, we provide a couple of constants and lemmas for the new state. We start with defining the domain, the range and the empty state. The first two definitions are straightforward projections from the original map

constdefs
  $dom' :: '\alpha\ state \Rightarrow oid\ set$
  $dom'\ \sigma \equiv dom\ (Rep\_state\ \sigma)$

constdefs
  $ran' :: '\alpha\ state \Rightarrow ('\alpha\ U)\ set$
  $ran'\ \sigma \equiv ran\ (Rep\_state\ \sigma)$

constdefs
  $mt\_state :: '\alpha\ state$
  $mt\_state \equiv Abs\_state\ (\lambda\ x.\ None)$

Memory

As IMP++ will support object creations, the state has to support an operation which creates new objects. Such an object creation will only complete successfully, if there is enough free memory to store the new object. We model the free memory as having at least one more oid which is not used yet. This is sensible, because if there was no more oid available, we could not create the object.

There are two possibilities to formalize the requirement that the set of non-used but defined oids is not empty. Either we assume a finite memory and require that all subsequent object creations will succeed because there is enough space, or we exploit the fact that all partially correct programs can only consume a finite amount of memory while we assume that the memory is infinite.

constdefs
$\mathit{enough\_space\_for} :: {}'\alpha\ \mathit{state} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$
$\mathit{enough\_space\_for}\ \sigma\ k \equiv (\exists\ s.\ ((s \subseteq (-\ (\mathit{dom}'\ \sigma \cup\ \{\mathit{NULL}\}))) \wedge \mathit{card}\ s = k \wedge \mathit{finite}\ s))$

This is the first variant. It says that there is a finite subset of the universe of all oids minus the ones which are already used and the undefined one, whose cardinality is k.

constdefs
$\mathit{infinite\_memory} ::\ \mathit{bool}$
$\mathit{infinite\_memory} \equiv \neg\ \mathit{finite}\ (\mathit{UNIV} : :\ \mathit{oid\ set})$

This definition is for the second variant. It says that the set of all oids is infinite.

Both choices are valid options leading to a program verification method. We therefore provide another definition, *memory_not_full*, which simply says that there is at least one oid available. All subsequent lemmas about object creation will be working with this lemma. This way, we can use both options later in program verification

constdefs
$\mathit{memory\_not\_full} :: {}'\alpha\ \mathit{state} \Rightarrow \mathit{bool}$
$\mathit{memory\_not\_full}\ \sigma \equiv \mathit{dom}'(\sigma) \subset \mathit{UNIV} - \{\mathit{NULL}\}$

We then need the following two lemmas which provide the transition between the definitions:

lemma *enough_space_implies_not_full*:
$[\![\mathit{enough\_space\_for}\ \sigma\ k;\ k > 0]\!] \Longrightarrow \mathit{memory\_not\_full}\ \sigma$

lemma *infinite_memory_implies_not_full*:
$[\![\mathit{infinite\_memory};\ \mathit{finite}\ (\mathit{dom}'\ \sigma)]\!] \Longrightarrow \mathit{memory\_not\_full}\ \sigma$

The state has to support three basic operations: the creation of new objects in the state, the update of an existing object, and the access of an object in the state. These operations will be presented in the following.

Create an object in the state

Before we give the definition of the create operation, we outline the conditions we impose on such a definition. Its type should be a state transition, where the returned state is equal to the old one but with one additional element. As we want to be able to access the new object later, the operation must also return its oid. Furthermore, the operation must satisfy the following four conditions. These conditions can also be seen as the characterisation of the create operation.

1. The new oid must not be in the domain of the old state.
2. The new oid must not be *NULL*.
3. The domain of the new state is the old domain with the new oid added.
4. All objects in the old state remain untouched.

To specify a new oid, we can use the Hilbert operator for indefinite descriptions, written as $\varepsilon\ x.\ P\ x$. It returns some x such that $P\ x$ is true, provided one exists.
The following definition fulfills all these conditions:

constdefs
$create\_OclAny\_in\_state :: {'\alpha}\ state \Rightarrow oid \times {'\alpha}\ state$
$create\_OclAny\_in\_state\ \sigma \equiv$
$let\ new = \epsilon\ x.\ (x \neq NULL \wedge x \notin (dom'\ \sigma) \wedge dom'(\sigma) \subset UNIV - \{NULL\})$
$in\ (new,\ Abs\_state(Rep\_state(\sigma)$
$(new \mapsto (level0.oclLib.mk\_OclAny\ \llcorner((OCL\_OclAny\_type.OclAny, new), \bot)\lrcorner))))$

The part $dom'(\sigma) \subset UNIV - \{NULL\}$ states that there is at least one element available which is neither *NULL* nor in the domain. It is equivalent to the *memory_not_full* predicate. Putting it inside the predicate for the Hilbert operator is technically not necessary, but facilitates later proof work.
The definition has to be checked against the conditions. The corresponding lemmas are proved as *create_charn1* to *create_charn4* in the state theory.

We need another crucial lemma: the memory will be filled by not more than one additional object during a create operation. The lemma looks pretty trivial, however its proof is rather involved as the Isabelle library for the cardinality of sets is not well developed yet.

lemma *enough_space_for_create*:
$[\![k > 0;\ enough\_space\_for\ \sigma\ k]\!] \Longrightarrow$
$enough\_space\_for\ (snd\ (create\_OclAny\_in\_state\ \sigma))\ (k - (1 :: nat))$

Update an object in the state

The next operation we need is an update. As the create, the update is basically a state transition. Additionally, we need to know which object we want to update, and its updated value. As the first component of an object, (*OclAny_key*, *oid*), must never change, we do not include the whole updated object as argument, but rather only the other part, called extension.

Again, there are a number of conditions on an update operation which hold whenever the operation succeeds. These are:

1. The domain remains unchanged.
2. The oid element of the updated tuple is still the same.
3. The extension of the updated element is what we supplied to the update operation.
4. All other objects remain untouched.
5. A second update on the same object cancels the former.
6. The update operation is commutative if invoked on different objects.

The following definition satisfies these conditions:

```
constdefs
  updext_OclAny_in_state :: [oid, 'α up] ⇒ 'α state ⇒ 'α state
  updext_OclAny_in_state oid x σ ≡
    if    oid ∈ dom' σ ∧ level0.oclLib.is_OclAny_univ(the(Rep_state σ oid))
    then  let new = (level0.oclLib.mk_OclAny ⌊((OCL_OclAny_type.OclAny,oid),x)⌋)
          in  (Abs_state(Rep_state(σ) (oid ↦ new )))
    else  arbitrary
```

The lemmas that the conditions are indeed fulfilled are proved as *updext_charn1* to *updext_charn_4*, *updext_cancel*, and *updext_commute*.

Access an object in the state

The last one of the basic operations is the access. Given a state and an oid, it should return the corresponding object.

The conditions on this operation are:

1. The access on an updated element should return the updated object.
2. The access on an element should return the same object, even if other objects have been updated in between.

```
constdefs
  access_OclAny_in_state :: ['α state, oid] ⇒ 'α OclAny_0
  access_OclAny_in_state σ oid ≡
     if   oid ∈ dom' σ ∧ level0.oclLib.is_OclAny_univ (the (Rep_state σ oid))
     then level0.oclLib.get_OclAny (the (Rep_state σ oid))
     else arbitrary
```

The lemmas for the compliance of this definition to the conditions are proved under the names *access_charn1* and *access_charn2* in the state theory.

Underspecification

All the three operations we presented are underspecified. The create operation returns an undefined oid if there is no more available in the memory, the update returns arbitrary if the element to be updated was either not in the state or of a wrong type, and analogously for the access operation. This means that on this level we have to work with explicit side conditions. On level 1 these will generally be transformed to undefinedness calculations.

A Calculus for the State

Although we now have everything essential together, more lemmas are necessary if we want to ensure effective and efficient proving over states. We provide a big amount of generic lemmas which provide a calculus for reasoning over states. These lemmas are inspired from the analogous lemmas for the map in Isabelle, but go further. We won't provide a complete overview over all the lemmas at this point, but rather pick out a few interesting ones which are used regularly later. We present the lemmas without their proofs here. They are of course all proved in the corresponding Isabelle theory files.

We start with giving the definition of a predicate we will make heavy use of later. Very often we will encounter assumptions of the kind of "if the object is in the state and of a specific type, then ...". For example we use this kind of statement in the conditions of the access and update operations. To simplify later proof work, we introduce a predicate which is formalizing this statement. When we will later in the thesis write informal statements of the form "a correct object in the state", we actually mean that the respective predicate, which will be defined for each class of the model, evaluates to true.

constdefs
object_in_state $::'\alpha$ *state* $\Rightarrow$ *oid* $\Rightarrow$ *bool*
object_in_state σ *oid* $\equiv$ *oid* $\in$ *dom'* σ $\wedge$
level0.oclLib.is_OclAny_univ (*the* ((*Rep_state* σ) *oid*))

It will sometimes be necessary to unfold the definition of the access or update operations. If we know that *object_in_state* is true, we know that also the conditions in the definitions are true. The following two lemmas shortcut the unfolding in those cases. The way the lemmas are written makes them easy to apply together with the simplifier or as a substitution rule. These methods work such that they replace the occurrence of the left hand side of the lemma in a subgoal with the right hand side. The *simp* rewrites every occurrence, the *subst* only one of them. Which one can be changed with the *back* command.

lemma *access_unfold*:
object_in_state σ *oid* $\Longrightarrow$
access_OclAny_in_state σ *oid* = *level0.oclLib.get_OclAny* (*the* (*Rep_state* σ *oid*))

lemma *updext_unfold*:
object_in_state σ *oid* $\Longrightarrow$ *updext_OclAny_in_state oid foo* σ =
(*Abs_state* (*Rep_state* σ(*oid* $\mapsto$ *level0.oclLib.mk_OclAny* $\lfloor$((*OclAny_key.OclAny*, *oid*), *foo*)$\rfloor$)))

The following lemma returns the oid of an object we just accessed in the state. From the state invariant we know that this oid must be the one which we used for the access. The proof is basically an unfolding of the access operation and the state invariant, followed by an application of the simplifier.

lemma *oidOfX*:
$[\![$*object_in_state* σ *self_oid*; *access_OclAny_in_state* σ *self_oid* $= x]\!] \Longrightarrow$
OclOidOf0 $x =$ *self_oid*

If we want to reason about an object we have just created or just want to state that the creation operation went well, we need to know that the new oid is not *NULL*. So we need to state that the result of the Hilbert operator which is used for the creation of a new oid is not *NULL*. This fact is stated in the following lemma. To prove it, we must prove that there is at least one x which satisfies the conditions of the Hilbert operator. This is exactly the case if the memory is not full yet.

lemma *new_oid_not_NULL*:
memory_not_full $\sigma \Longrightarrow$
$NULL \neq (\epsilon\ x.\ x \neq NULL \wedge x \notin (dom'\ \sigma) \wedge dom'(\sigma) \subset UNIV - \{NULL\})$

The following lemma is the analogue of an important corresponding lemma in the theory for maps in Isabelle. It states that the state we get from updating an object with its current value will remain the same.

lemma *state_upd_triv_access*:
$[\![$*object_in_state* σ *oid*;*access_OclAny_in_state* σ *oid* $= (\llcorner((OclAny\_key.OclAny,\ oid),y)\lrcorner)]\!]$
$\Longrightarrow$ *updext_OclAny_in_state oid y* $\sigma = \sigma$

We will often encounter operations to access an object we have just updated. With the following lemma, we can rewrite this directly for the cases where the corresponding object is actually in the state and of correct type.

lemma *access_update*:
$[\![$*object_in_state* σ *oid*$]\!] \Longrightarrow$
access_OclAny_in_state (*updext_OclAny_in_state oid x* σ) *oid* $=$
$\llcorner((OclAny\_key.OclAny,\ oid),\ x)\lrcorner$

The following lemma is a variation of number four of the characterisations of the update operation. An object does not change if we update another one. Here the statement is formulated in a way which makes it more amenable to rewriting.

lemma *access_update_other*:
$[\![$*object_in_state* σ *oid*; *object_in_state* σ *other_oid*; *other_oid* $\neq$ *oid*$]\!] \Longrightarrow$
access_OclAny_in_state (*updext_OclAny_in_state oid foo* σ) *other_oid* $=$
access_OclAny_in_state σ *other_oid*

The next lemma, and variations thereof, will be used very heavily later. Its task is to propagate the *object_in_state* predicate to an updated state. We know from the first condition we have set upon the update operation, that the domain must remain the same. As we also ensured that an oid always points to the same object, which can only be updated in a restricted way with the update operation, we can also infer that it must still have the same type. Thus the predicate still holds in an updated state. The lemma is true no matter if *oid* and *other_ oid* are different or not.

lemma *object_ in_ state_ update*:
$[\![\textit{object\_in\_state}\ \sigma\ \textit{other\_oid};\ \textit{object\_in\_state}\ \sigma\ \textit{oid}]\!] \Longrightarrow$
$\textit{object\_in\_state}\ (\textit{updext\_OclAny\_in\_state oid x}\ \sigma)\ \textit{other\_oid}$

And now the same statement for the create operation. We need to assure that this operation went well, which is assured by the *memory_ not_ full* predicate. Otherwise the new state would be arbitrary and we would not be able to prove the statement.

lemma *object_ in_ state_ create*:
$[\![\textit{object\_in\_state}\ \sigma\ \textit{oid};\ \textit{memory\_not\_full}\ \sigma]\!] \Longrightarrow$
$\textit{object\_in\_state}\ (\textit{snd}\ (\textit{create\_OclAny\_in\_state}\ \sigma))\ \textit{oid}$

The next lemma states that an object does not change during an update. This is already assured by the second characterisation of the update operation, but here formulated in a way which makes it more useful for proving.

lemma *oid_ of_ update*:
$[\![\textit{object\_in\_state}\ \sigma\ \textit{oid};\ \textit{post} = \textit{updext\_OclAny\_in\_state oid foo}\ \sigma]\!] \Longrightarrow$
$\textit{OclOidOf0}\ (\textit{access\_OclAny\_in\_state}\ \sigma\ \textit{oid}) = \textit{OclOidOf0}\ (\textit{access\_OclAny\_in\_state post oid})$

Two oids must never denote the same object. This is a consequence from condition number three to the state, which asserts a one-to-one correspondence between oids and objects.

lemma *obj_ not_ eq*:
$[\![\textit{object\_in\_state}\ \sigma\ \textit{oid};\ \textit{object\_in\_state}\ \sigma\ \textit{oid}';\ \textit{oid} \neq \textit{oid}']\!] \Longrightarrow$
$\textit{Rep\_state}\ \sigma\ \textit{oid} \neq \textit{Rep\_state}\ \sigma\ \textit{oid}'$

State for the Example Model

The state is a mapping from oid to *'a Universe*. This universe is model-specific, the encoder of HOL-OCL creates it according to the classes in the model. The same is true for the state, which is basically a mapping from oids to this universe. The state presented before, *'a state*, is the generic one. When we reason about a specific model or program, the "hole" *'a* is filled by the corresponding type.

In the following, we present the state for the example with the two classes `Node` and `Cnode` we presented in Chapter 2. The type of this state is as follows. For the moment, the StackObject part can be ignored, it will be introduced later.

types $('\alpha,'\beta,'\gamma,\ '\delta)\ \textit{list\_state} = ((\textit{Node\_key} \times \textit{Integer\_0} \times \textit{oid}) \times$
$((\textit{Cnode\_key} \times \textit{Boolean\_0}) \times {'\alpha}\ \textit{up} + {'\beta})\ \textit{up} +$
$(\textit{StackObject\_key} \times \textit{oid} \times \textit{oid} \times \textit{oid}) \times {'\gamma}\ \textit{up} + {'\delta})\ \textit{state}$

The three state operations, the *object_ in_ state* predicate, and all the generic lemmas were defined for `OclAny` objects. We now need to do the same for every class type in our model. This formalization is made in a modular way, which is already employed heavily in the HOL-OCL library. The idea behind this approach is that we can use the corresponding lemma of the parent class when proving a lemma for a specific class. This way, the whole method will scale and remain feasible for large models.

The only change in the definition of the create operation is the specification of the new object we create. The attributes of the new object remain undefined and are set to ⊥. Details of the encoding of the single objects will be presented in the next Section.

constdefs
```
  create_Node_in_state :: ('α,'β,'γ,'δ) list_state ⇒ oid × ('α,'β,'γ,'δ) list_state
  create_Node_in_state σ ≡
  let new = ϵ x. x ≠ NULL ∧ x ∉ dom' σ ∧ dom' σ ⊂ UNIV − {NULL}
  in (new, Abs_state (Rep_state σ(new ↦
      level0.list.mk_Node (⌊((OclAny_key.OclAny, new), ⌊((Node_key.Node, ⊥ , ⊥ ),⊥ )⌋)⌋))))
```

constdefs
```
  create_Cnode_in_state :: ('α,'β,'γ,'δ) list_state ⇒ oid × ('α,'β,'γ,'δ) list_state
  create_Cnode_in_state σ ≡
  let new = ϵ x. x ≠ NULL ∧ x ∉ dom' σ ∧ dom' σ ⊂ UNIV − {NULL}
  in (new, Abs_state (Rep_state σ(new↦
      level0.list.mk_Cnode (⌊((OclAny_key.OclAny, new), ⌊((Node_key.Node, ⊥ , ⊥),
                           ⌊((Cnode_key.Cnode, ⊥),⊥)⌋ )⌋)⌋))))
```

The access and update operations have to test if the desired object is of correct type.

constdefs
```
  access_Node_in_state :: [('α,'β,'γ,'δ) list_state, oid] ⇒ ('α,'β) Node
  access_Node_in_state σ oid ≡
    if   oid ∈ dom' σ ∧ level0.list.is_Node_univ(the(Rep_state σ oid))
    then level0.list.get_Node (the (Rep_state σ oid))
    else  arbitrary
```

constdefs
```
  access_Cnode_in_state :: [('α,'β,'γ,'δ) list_state, oid] ⇒ 'α Cnode
  access_Cnode_in_state σ oid ≡
    if   oid ∈ dom' σ ∧ level0.list.is_Cnode_univ (the (Rep_state σ oid))
    then level0.list.get_Cnode (the(Rep_state σ oid))
    else  arbitrary
```

constdefs
```
  updext_Node_in_state ::
  [oid, ((Node_key × Integer_0 × oid) × ((Cnode_key × Boolean_0) ×  'α up + 'β)up) up] ⇒
   ('α,'β,'γ,'δ) list_state ⇒ ('α,'β,'γ,'δ) list_state
  updext_Node_in_state oid x σ  ≡
    if   level0.list.is_Node_univ (the ((Rep_state σ) oid))
    then updext_OclAny_in_state oid (⌊Inl ⌜x⌝⌋) σ
    else  arbitrary
```

constdefs
updext_Cnode_in_state ::
[*oid*, ((*Node_key* $\times$ *Integer_0* $\times$ *oid*) $\times$ ((*Cnode_key* $\times$ *Boolean_0*) $\times$ *'α up*)*up*) *up*] $\Rightarrow$
('α,'β,'γ,'δ) *list_state* $\Rightarrow$ ('α,'β,'γ,'δ) *list_state*
updext_Cnode_in_state oid x σ $\equiv$
  if level0.list.is_Cnode_univ (*the* ((*Rep_state* σ) *oid*))
  then updext_Node_in_state oid $\lfloor$(*sup x*, $\lfloor$*Inl* $\ulcorner$(*base x*)$\urcorner\rfloor$)$\rfloor$ σ
  else arbitrary

Of course, all the characterisation lemmas are proved for these definitions as well.

We also add analogues of the *object_in_state* predicate for each class.

constdefs
Node_in_state ::('α,'β,'γ,'δ) *list_state* $\Rightarrow$ *oid* $\Rightarrow$ *bool*
Node_in_state σ *oid* $\equiv$ *oid* $\in$ *dom'* σ $\wedge$ *level0.list.is_Node_univ* (*the* ((*Rep_state* σ) *oid*))

constdefs
Cnode_in_state ::('α,'β,'γ,'δ) *list_state* $\Rightarrow$ *oid* $\Rightarrow$ *bool*
Cnode_in_state σ *oid* $\equiv$ *oid* $\in$ *dom'* σ $\wedge$ *level0.list.is_Cnode_univ* (*the* ((*Rep_state* σ) *oid*))

We then need to state that all the generic lemmas we have proved for `OclAny` also hold for the `Node` and `Cnode` classes. For being able to conduct proofs using the modular approach just outlined, we need some lemmas about the connection between the classes. They always work from one level to the next, not further. There is for example no lemma about the connection between `OclAny` and `Cnode`, this can be achieved with a combination from `OclAny` to `Node` and from `Node` to `Cnode`. This way the number of lemmas won't increase exponentially.

The following connection lemma says that if we know that *Node_in_state oid* σ is true, also *object_in_state oid* σ must be true. This holds as every `Node` object is also an `OclAny` object.

lemma *Node_object_in_state*:
Node_in_state oid σ $\Longrightarrow$ *object_in_state oid* σ

Next is the connection lemma for the update operation, but this time the other way round. Of course this lemma only holds if the object is of `Node` type.

lemma *updext_OclAny_Node*:
$[\![$*Node_in_state s oid*; *P* (*updext_OclAny_in_state oid* ($\lfloor$*Inl* $\ulcorner$*foo*$\urcorner\rfloor$) *s*)$]\!]$ $\Longrightarrow$
P (*updext_Node_in_state oid foo s*)

We can now give an example to show how the modular system works in practice. We have already encountered the lemma *object_in_state_update* in the previous section. Now we want to prove the same for the `Node` object. Rule applications number one, three and five in the following proof are connection lemmas, *upd_is_Node_univ_aux* is a small auxiliary lemma about `Nodes`. Rule application number four then applies the corresponding lemma of the parent class.

```
lemma Node_in_state_update:
  Node_in_state σ oid ⟹ Node_in_state (updext_Node_in_state oid x σ) oid
   apply (rule object_Node_in_state)
   apply (erule upd_is_Node_univ_aux)
   apply (rule updext_OclAny_Node, assumption)
   apply (rule object_in_state_update)
   apply (erule Node_object_in_state)
done
```

And this is the proof for the same lemma for `Cnode`.

```
lemma Cnode_in_state_update:
  Cnode_in_state σ oid ⟹ Cnode_in_state (updext_Cnode_in_state oid x σ) oid
   apply (rule Node_Cnode_in_state)
   apply (erule upd_is_Cnode_univ_aux)
   apply (rule updext_Node_Cnode, assumption)
   apply (rule Node_in_state_update)
   apply (erule Cnode_Node_in_state)
done
```

We can clearly see a pattern. It is exactly the same proof as for the `Node`, simply one class further down. For the proof for `Cnode`, we can reuse the proof script of the same lemma for `Node`, and just change the lemmas we apply to the ones one level further down. The modular approach always works this way. For proving a lemma for a class on level i in the class hierarchy, we only use lemmas of levels i and $i-1$, but never go further up. This leads to proofs which are completely analogous, independent of the size of the model. The lemma we just presented would be exactly the same size for a child or grand-child of `Cnode`. Basically, we could just do a global search and replace for the theorem names.

Most lemmas are proved with the approach just presented. There are of course also lemmas which can be proved directly and uniformly on the level of the same class. We should however avoid the unfolding of definitions of parent classes in a proof, or go further up the class hierarchy than just one step. This leads to a methodology which also scales for large class models.

4.2 Level 0

Level 0 provides the encoding of single objects of an object-oriented model. The encoder provides class types, a universe type, type casts, etc. The way these objects are modelled can be reused completely for the context of program verification. We thus do not have to change much on this level. The definitions which are already being created work still well. The only essential things which are missing are the setters for the attributes.

An object is basically stored as a tuple. Details of how this tuple gets created are given in [2]. As an example, we show an example of one specific `Cnode` tuple:

$\llcorner((OclAny\_key,\ oid),\ \llcorner((Node\_key,\ i,\ next),\ \llcorner((Cnode\_key,\ color),\bot)\lrcorner)\lrcorner)\lrcorner$

OclAny_key, Node_key, and *Cnode_key* are tag types, *oid* is the object identifier, *i* is the `i` attribute, *next* the oid of the next attribute, *color* the color attribute, and the element $\bot$ is the undefined extension. This extension would be defined if we inherit from `Cnode`. It should be noted that in the `next` attribute we just store the oid of the object it points to. The level 0 representation is therefore not type safe, as this oid could denote an object of any type. The type safety will be introduced on level 1.

The encoder provides getters to access the attributes of an object. In the context of a side-effect free specification language like UML/OCL, there are no setters for the attributes necessary. For a programming language semantics however, we need them. These are the only essential extensions to the object store of HOL-OCL.

There are not only getters for each of the attributes, but also for other parts of the tuple. For a `Cnode`, these include the following three operators, here given with the result if applied to the example above:

- *CnodeGetAttrs*: (*Cnode_key*, *color*)
- *CnodeGetNode*: (*Node_key*, *i*, *next*)
- *CnodeGetExtension*: $\bot$

We also need the following operator, which is heavily used on level 0. It is the counterpart of the fst operator over lifted tuples. If we apply this operator on the `Cnode` example above, it returns (*OclAny_key*, *oid*).
$sup\ obj \equiv fst\ \ulcorner obj \urcorner$

Analogously to the getters, which are projections out of a tuple, the setters have to overwrite the correct element in the tuple. We refrain from stating all the definitions, but only give an example, namely the definition of setting the `color` attribute of a `Cnode` object. The other setters are defined analogously.

```
constdefs
  SetColor :: 'α Cnode ⇒ bool up ⇒ 'α Cnode
  SetColor self up_color ≡ let attr = CnodeGetAttrs self in
                           let ext  = CnodeGetExtension self in
                           let node = CnodeGetNode self in
      ⌊(sup self, ⌊(node, ⌊((fst attr, up_color), ext)⌋)⌋)⌋
```

From this definition we see that the objects are not smashed. This means that a defined object may have undefined elements. This is in contrast to the usual smashing semantics of tuples and sets in OCL. However, it makes perfect sense in this context, e.g. the attributes of an object do not have to be defined at creation time. This is consistent with the behaviour of constructors in those programming languages where the attributes do not get initialized with a default value.

4.3 Level 1

With level 0 and the state being ready, we can go on to level 1. There are two major properties appearing on this level. At first we are now dealing explicitly with definedness and strictness. Second, the semantics of an expression now always depends on the current state. We are usually not interested in the value of a specific attribute, but rather in the value of a specific attribute in a specific context. Therefore we lift over the context any semantic function occurring in an expression. HOL-OCL provides several types and operators to automate this context-lifting.

First, there is the type synonym VAL with the meaning: $(\sigma, \alpha)\ VAL = (\sigma \Rightarrow \alpha)$

Then there are a couple of lifting operators. E.g:

constdefs
  lift0 :: $'\alpha \Rightarrow ('\tau,'\alpha)\ VAL$
  lift0 $\equiv \lambda c.\ \lambda s.\ c$

Analogously, there are operators *lift1*, *lift2*, *lift3* for context lifting of functions with one, two, or three arguments.

The following constant makes an operation strict:

constdefs
  strictify :: $('\alpha ::\ bot \Rightarrow '\beta ::\ bot) \Rightarrow '\alpha \Rightarrow '\beta$
  strictify f x $\equiv$ *if* $x{=}\bot$ *then* $\bot$ *else f x*

These and several other operators help to make the embedding modular and reduce the amounts of lemmas which have to be proved for the calculus. For this reason, we will use them as often as possible.

Unfortunately we can not reuse the level 1 formalisations from the existing HOL-OCL library. The reason is that the context in the former is a pair of states, namely the pre and and the post state. When dealing with a small step semantics on the other hand, we only have one state, the current one. While for some definitions only the type has to be adjusted and the lifting operators can be reused, other definitions change more. There are completely new definitions when it comes to the setters. They do not only have to update an object, but also have to access and update this corresponding object in the state.

Whenever we talk about an object *self* on this level, this *self* is a function from a state to a level 0 object. If we have expressions of the form "the attribute of *self* in some state is *foo*", we have an expression of the form (*self* σ). But we do not know which level 0 object this expression denotes, as we do not know much about the function *self*. *self* has to be characterised further to establish the correspondence between a level 0 and level 1 object.

This correspondence is very hard to get right, a wrong formalisation could easily make the model inconsistent. What we need in the end is a mapping between a level 1 identifier like *self* and an object identifier, with special care for the cases where this object does not exist in a specific state. We finally got to the following definition:

```
constdefs
  is_handle_for :: ['α state ⇒ 'α OclAny_0, oid] ⇒ bool
  is_handle_for self oid ≡ (∀ σ. ((oid ∈ dom' σ) ∧
                            (level0.oclLib.is_OclAny_univ (the (Rep_state σ oid)) )) ⟶
                            (self σ) = access_OclAny_in_state_new σ oid)
```

This predicate establishes a clear correspondence between a level 1 object and its identifier, as long as it is defined and in the state. The motivation comes from the standard object-oriented assumption that an object identifier never changes during its life time, and that *self* should project exactly on the tuple with the correct oid.

An important sanity check for our definition is to assure that such a handle does actually exist, otherwise adding this predicate as assumption to a proof would introduce an inconsistency.

```
lemma handle_exists: ∃ self. is_handle_for self oid
  apply (simp add: is_handle_for_def access_OclAny_in_state_new_def)
  apply (rule_tac x = λ σ. level0.oclLib.get_OclAny (the (Rep_state σ oid)) in exI)
  apply (simp)
done
```

One of the first definitions we need is a level 1 oid. This definition uses the provided lifting combinators:

```
constdefs
  l1_OclOid :: ('α state, 'β OclAny_0) VAL ⇒ ('α state, oid) VAL
  l1_OclOid ≡ lift1 (strictify OclOidOf0)
```

The *strictify* operator ensures that the oid of an undefined object will not be defined.

We can already prove the following essential lemma, which states that the level 1 oid of an object is the one which is specified as its handle.

```
lemma l1_oid_is:
 ⟦is_handle_for self oid; object_in_state σ oid⟧ ⟹
  l1_OclOid self σ = oid
```

Which immediately leads to the following important fact, ensuring that a level 1 oid will not change during normal program execution:

```
lemma l1_oid_of_update:
 ⟦is_handle_for self oid; object_in_state σ oid; post = updext_OclAny_in_state oid foo σ⟧ ⟹
  l1_OclOid self σ = l1_OclOid self post
```

We now present the type-safe level 1 encoding of the objects and operators for our example.

At first, we need to have a handle for `Node` and `Cnode` objects. Also for these definitions, existential proofs are provided.

constdefs

$$\begin{array}{l}
\quad is\_handle\_for\_Node :: [('\alpha,\ '\beta,\ '\gamma,\ '\delta)\ list\_state \Rightarrow ('\alpha,\ '\beta)\ Node,\ oid] \Rightarrow bool \\
\quad is\_handle\_for\_Node\ self\ oid \equiv \\
\qquad (\forall\ \sigma.(((oid \in dom'\ \sigma) \wedge (level0.list.is\_Node\_univ\ (the\ (Rep\_state\ \sigma\ oid))) \longrightarrow \\
\qquad (self\ \sigma) = access\_Node\_in\_state\ \sigma\ oid)))
\end{array}$$

The following operator takes a state and an oid, and returns the context lifted `Node`, if it exists.

constdefs

$$\begin{array}{l}
\quad l1\_get\_Node :: ('\alpha,'\beta,'\gamma,'\delta)\ list\_state \Rightarrow oid \Rightarrow (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL \\
\quad l1\_get\_Node\ \sigma\ oid \equiv if\quad Node\_in\_state\ \sigma\ oid \\
\qquad\qquad\qquad\qquad\qquad then\ (lift0\ (access\_Node\_in\_state\ \sigma\ oid)) \\
\qquad\qquad\qquad\qquad\qquad else\ \bot
\end{array}$$

The following two definitions of getters are straightforward liftings of their level 0 counterparts.

constdefs

$$\begin{array}{l}
\quad l1\_Node\_get\_ext :: (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL \Rightarrow \\
\qquad (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ((Node\_key \times int\ up \times oid) \times \\
\qquad ((Cnode\_key \times bool\ up) \times '\alpha\ up + '\beta)\ up)\ up)\ VAL \\
\quad l1\_Node\_get\_ext \equiv lift1\ (strictify\ Nodegetext)
\end{array}$$

constdefs

$$\begin{array}{l}
\quad l1\_Node\_extension :: (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL \Rightarrow \\
\qquad (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ((Cnode\_key \times bool\ up) \times '\alpha\ up + '\beta)\ up)\ VAL \\
\quad l1\_Node\_extension\ \equiv lift1\ (strictify\ Nodeextension)
\end{array}$$

We can now define the getters fo the attributes on level 1. The definitions differ if the attributes are of value or of object type. For value types like `size`, the definition is the straightforward lifting.

constdefs

$$\begin{array}{l}
\quad l1\_Node\_size :: (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL \quad \Rightarrow \\
\qquad\qquad\qquad (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ int\ up)\ VAL \\
\quad l1\_Node\_size\ \equiv\ lift1\ (strictify\ level0.list.Node.size)
\end{array}$$

Things are a little more complicated for attributes of object type. These are represented as their oid on level 0. On level 1 however, it would not make sense to define them as level 1 oid. We need to ensure type-safety on this level, i.e. there can only be an object of type Node as the `next` attribute. Therefore the getter for `next` returns a level 1 `Node`, but only if it exists.

constdefs

$$\begin{array}{l}
\quad l1\_Node\_next :: (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL \quad \Rightarrow \\
\qquad\qquad\qquad (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL \\
\quad l1\_Node\_next \equiv \lambda X\ \sigma.\ if\ Node\_in\_state\ \sigma\ (level0.list.Node.next\ (X\ \sigma)) \\
\qquad\qquad\qquad\qquad\qquad then\ access\_Node\_in\_state\ \sigma\ (level0.list.Node.next\ (X\ \sigma)) \\
\qquad\qquad\qquad\qquad\qquad else\ \bot
\end{array}$$

The setters are basically transitions on states. Their outermost type is *state* ⇒ *state up*. They are allowed to return the undefined state. Additionally, the setter must know which object we want to update, and the value of the attribute to be updated. In the definition, we need to get the level 0 representations of the oid, the attribute, and the object, and then make an update into the state, with the argument given being the result of the level 0 setter. If the object we want to update is either not in the state or not of correct type, the state to be returned is undefined, and the result is ⊥. This way we can model exceptions. We provide the two examples for the size and the next attribute.

```
constdefs
 l1_Node_set_size :: (('α,'β,'γ,'δ)  list_state, ('α,'β) Node) VAL
                        ⇒ (('α,'β,'γ,'δ) list_state, int up) VAL
                         ⇒ ('α,'β,'γ,'δ) list_state
                         ⇒ ('α,'β,'γ,'δ) list_state up
 l1_Node_set_size self up_s σ ≡
        let oid =    (l1_OclOid self) σ in
        let new_s = up_s σ in
        let self_0 =  access_Node_in_state σ oid in
 if   Node_in_state σ oid
 then ⌊(updext_Node_in_state oid (Nodegetext (Setsize (self_0) (new_s))) σ )⌋
 else ⊥
```

```
constdefs
 l1_Node_set_next :: (('α,'β,'γ,'δ)  list_state, ('α,'β) Node) VAL
                        ⇒ (('α,'β,'γ,'δ) list_state, ('α,'β) Node) VAL
                        ⇒ ('α,'β,'γ,'δ) list_state
                        ⇒ ('α,'β,'γ,'δ) list_state up
 l1_Node_set_next self up_next σ ≡
         let oid       =  (l1_OclOid self) σ in
         let self_0   =  access_Node_in_state σ oid  in
         let next_oid = (l1_OclOid up_next) σ in
 if   Node_in_state σ oid
 then ⌊(updext_Node_in_state oid (Nodegetext (Setnext self_0 next_oid))  σ )⌋
 else ⊥
```

What happens in this definition if the *up_next* object is not in the state? As *l1_OclOid* is defined as strict, *next_oid*, the value of the `next` attribute, will be set to ⊥, but the operation would return a correct state. This is coherent to our decision to give objects a non-smashed semantics. However, the necessary changes if one would like the operation to return the ⊥ state in that case, would be rather small.

It is important to see why with such a definition, type-safety is ensured implicitly. It is impossible to set the `next` attribute to anything other than something of type `Node`, as then the arguments supplied to the operation would be wrongly typed, in which case Isabelle would report. If we would like to supply a `Cnode`, which is of course legal in an object-oriented setting, we will have to cast it to `Node` before.

As we can not create new objects out of nowhere, we include another type of setters, the so called update_news. These operators update an object attribute with a newly created element. As an example, here is the definition for the update_new of the next attribute of a `Node`.

constdefs
$l1\_Node\_update\_next\_new :: (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL \Rightarrow$
$('\alpha,'\beta,'\gamma,'\delta)\ list\_state \Rightarrow$
$('\alpha,'\beta,'\gamma,'\delta)\ list\_state\ up$
$l1\_Node\_update\_next\_new\ self\ \sigma \equiv if\ memory\_not\_full\ \sigma\ then$
$let\ oid = (l1\_OclOid\ self)\ \sigma\ in$
$let\ (new\_oid,new\_state) = create\_Node\_in\_state\ \sigma\ in$
$let\ new\_Node = lift0\ \llcorner((OclAny\_key.OclAny,\ new\_oid),\llcorner((Node\_key.Node,\ \perp,\ \perp),\ \perp)\lrcorner)\lrcorner\ in$
$if\ \ Node\_in\_state\ \sigma\ oid$
$then\ l1\_Node\_set\_next\ self\ new\_Node\ new\_state$
$else\ \perp\ else\ \perp$

We need different kinds of definitions for each of the possible subclasses for the attribute in question. E.g. there exists also a definition to update_new a `Cnode` in the `next` attribute. The operators return the $\perp$ state if either the memory is full or the object to be updated is not a correct member of the state. The attributes of the newly created object always remain uninitialised.

The last type of definitions we have to add are type casts. They already exist on level 0 and can be lifted to level 1 using the standard operators. As an example, we state the definition of a cast from a `Cnode` to a `Node` object

constdefs
$l1\_Cnode\_2\_Node :: (('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ '\alpha\ Cnode)\ VAL \Rightarrow$
$(('\alpha,'\beta,'\gamma,'\delta)\ list\_state,\ ('\alpha,'\beta)\ Node)\ VAL$
$l1\_Cnode\_2\_Node \equiv lift1\ level0.list.Cnode\_2\_Node$

4.4 Level 2

The level 1 of HOL-OCL and the one of IMP++ are completely separated. On level 2, the support for invariants, preconditions, and postconditions is added. It is on this level where we can establish the interfacing between reasoning over the specification and over the implementation.
The main idea behind the interfacing is, that we specify the invariant using OCL and provide an encoding of this invariant. Using the Hoare calculus, we can establish that our implementation satisfies the invariant as specified in the specification, and therefore that the implementation conforms to the specification.

Let's recall the invariant as it gets specified in OCL:

```
context Cnode
inv flip:  self.color <> self.next.color
```

The encoding of the invariant is the following definition:

```
constdefs
 Cnode_inv_enc :: ('α,'β,'γ,'δ) list_state ⇒ ('α,'β) Node set
 Cnode_inv_enc ≡ ( λ σ . (λ f. (f σ)) ‘ (gfp (λC. {obj.
          (((σ ⊨ (∂  (l1_Node_next  obj))) ∧
          ((σ ⊨  (∂ (l1_Cnode_color  (l1_Node_2_Cnode obj)))) ∧
          (σ ⊨ ((l1_Cnode_color (l1_Node_2_Cnode obj)) ‘<>‘ (
              l1_Cnode_color (l1_Node_2_Cnode (l1_Node_next obj))))))) ∧
          ((σ ⊨ (∂̸ ( (l1_Node_next obj)))) ∨ ((l1_Node_next obj σ) ∈
                             ((λf. (f σ)) ‘ (C)))))})))
```

This rather complicated looking definition will be created automatically by the final encoder when loading a UML/OCL model. As we are talking about cyclic object structures, it includes a greatest fixpoint over all `Cnode` objects which fulfil the invariant. The details about the construction can be found in [2].

The definition above takes a state and returns a set of objects. A specific object satisfies the invariant if it is a member of this set. Using the following definition, we can state directly that a specific object satisfies the invariant.

```
constdefs
 Cnode_inv :: ('α, 'β) Node ⇒ ('α ,'β, 'γ, 'δ) list_state ⇒ bool
 Cnode_inv self σ ≡ self ∈ Cnode_inv_enc σ
```

4.5 Encoding a Program

In this Section, we present how we can encode a program in IMP++. The idea is to provide a syntax which is close to a real object-oriented programming language. In the long term, it could be envisaged that an encoder will transform a code fragment, which could for example be written in Java, into IMP++ automatically.

Stack Object

If we want to verify that the body of a method indeed satisfies the specification of this method, we need to extend the class model with the stack object of the method. This stack object contains as attributes all local variables of the method. We will then assume that this stack object is defined upon entering the method. This reflects the stack behaviour of the operational semantics for methods. The term stack object is also used in the Java language specification.

The stack object has to support a large subset of the operations of a common object, e.g. access and update of attributes. If we add it as a class to the model, all these operations will as well be created automatically once the encoder is finalised. This is the reason why the stack object is a part of the type of the state and the universe.

The method of our example has three local variables: *N1*, *N2*, and *return*. They are modelled as attributes of the stack object. Therefore, the stack object of our example has three attributes of type `Cnode`.

Syntax Translations

Before we formalise the method body of the example, some pretty printing is necessary to ensure readability. The mechanism for achieving this in Isabelle is called syntax-translation. First, using the **syntax** command, we introduce uninterpreted notational elements. The **translations** command then relates input forms to logical expressions. Using syntax-translations, expressions are immediately replaced by the definition upon parsing, but can be made much more readable. With this mechanism, we aim to make the syntax of our language as close to a traditional programming language as possible.

As an example of how this mechanism works, we show the syntax-translations for the getter, the setter, and the update_new of the `next` attribute.

syntax
_next :: $(('\alpha,'\beta,'\gamma,'\delta)$ *list_state*, $'\beta$ *list*) *VAL* $\Rightarrow$ $(('\alpha,'\beta,'\gamma,'\delta)$ *list_state*, $('\alpha,'\beta)$ *Node*) *VAL* $\Rightarrow$ $'\delta$ *com* (_ *.next* *80*)

_next := :: $(('\alpha,'\beta,'\gamma,'\delta)$ *list_state*, $('\alpha,'\beta)$ *Node*) *VAL* $\Rightarrow$ $(('\alpha,'\beta,'\gamma,'\delta)$ *list_state*, $('\alpha,'\beta)$ *Node*) *VAL* $\Rightarrow$ $'\delta$ *com* (**infixl** *'.next* := *80*)

_new_next :: $(('\alpha,'\beta,'\gamma,'\delta)$ *list_state*, $('\alpha,'\beta)$ *Node*) *VAL* $\Rightarrow$ $'\delta$ *com* (_ *.new'_next* *80*)

translations
self .next $\rightleftharpoons$ (*l1_Node_next self*)
self .next := *a* $\rightleftharpoons$ *Cmd* (*l1_Node_set_next self a*)
self .new_next $\rightleftharpoons$ *Cmd* (*l1_Node_update_next_new self*)

We can new encode our program in IMP++.

constdefs
generate_cyclic_List :: $(('\alpha,'\beta,'\gamma,'\delta)$ *list_state*, $'\gamma$ *StackObject*) *VAL* $\Rightarrow$
    ((*Node_key* $\times$ *Integer_0* $\times$ *oid*) $\times$ ((*Cnode_key* $\times$ *Boolean_0*) $\times$ $'\alpha$ *up* + $'\beta$) *up* +
    (*StackObject_key* $\times$ *oid* $\times$ *oid* $\times$ *oid*) $\times$ $'\gamma$ *up* + $'\delta$) *com*

    generate_cyclic_List so $\equiv$

    so .n1 := *New*(*Cnode*) ;
    so .n2 := *New*(*Cnode*);
    so .cn1 .color := T;
    (*so .cn2*) *.color* := F;
    (*so .n1*) *.next* := (*so .n2*);
    (*so .n2*) *.next* := (*so .n1*);
    so .return := (*so .n1*)

In pseudo code, the program was:

```
N1 := New(Cnode);
N2 := New(Cnode);
N1 := N1.set_color true;
N2 := N2.set_color false;
```

```
N1 := N1.set_next N2;
N2 := N2.set_next N1;
return := N1;
```

We realise they look both very similar, with the exception that instead of only writing *n1*, we have to write *so .n1* as our local variables are technically attributes of the stack object. To make matters easier, we have even included some type casting in the syntax translations. The expression *so .cn1* returns the `N1` attribute of the stack object, after casting it from `Node` to `Cnode`.

Using the syntax translations, we make the reasoning over states implicit, they are hidden away. Looking at the type of the single commands, we realise they are all functions from *state* to *state up*, even if there are nested path expressions involved which are simulated by a sequence of accessor, update, and cast functions. The outermost type is always the same, and these commands fit perfectly into the *Cmd* slot provided by the definition of the syntax of IMP++.

The symbols T and F in the program are the definitions OclTrue and OclFalse. These are the context lifted Boolean operators extended by the bottom element $\perp$ of the HOL-OCL library. They are defined to have the same value in every state.

As the commands are all of type *Cmd*, it is rather easy to specify their Hoare rules. They are simply instantiations of the rule *cmd_hoare*. Let's look at an example, the other lemmas are analogue.

lemma *Node_set_next_hoare*:
$\models \{\lambda\ \sigma.\ \neg(\sigma \models err)\ \wedge (\ulcorner\sigma\urcorner \models (\partial\ (l1\_Node\_set\_next\ self\ a\ )))\wedge$
  $Q(l1\_Node\_set\_next\ self\ a\ \ulcorner\sigma\urcorner)\ \}$
    $self\ .next := a$
  $\{\lambda\ \sigma.\ \neg\ (\sigma \models err) \wedge Q\ \sigma\ \}$
 apply (*rule cmd_hoare*)
done

If an assertion Q holds in the state after executing the command, then Q must hold in the state before execution but with the next attribute set, if the setter is defined in that state.

5 Program Verification

In this Chapter we show how we can verify a program written in IMP++ using the extended version of HOL-OCL we presented in the previous Chapter. We first provide the calculus and then present a detailed proof that an invariant holds over the program that we presented before.

5.1 A Calculus for IMP++

The commands of our programming language are essentially transitions on states. If we reason about a program we therefore basically reason about state transitions. In this section, we provide a calculus over these state transitions which will enable proof support for reasoning over IMP++ programs.

The calculus consists of a large amount of lemmas, which, in the end, will all be created and proved automatically by the encoder. These lemmas are model-specific. They are different for each specific class model we are loading. However, they are independent from the specific program fragment or invariant we want to prove. They can be used for reasoning about any program written over the specific UML model.

Before we provide an overview over the calculus, we state one more definition. When reasoning over a program, we can assume that the stack object exists upon entering the method, and therefore can assume a handle for it as assumption of the lemma. For the objects which are created within the program however, like the two `Cnodes` in our example, a handle can not be assumed. This makes many lemmas more complicated. To ensure we can reuse the big majority of lemmas, we add the following definition for both classes of the model. This predicate takes an object, a state and an oid, and returns true if either the oid is a handle of the object, or if not, we can establish by some other means that in this specific state, the oid references exactly this object.

constdefs
$\mathit{is\_handle\_or\_oid} :: ((\prime\alpha,\prime\beta,\prime\gamma,\prime\delta)\ \mathit{list\_state},\ (\prime\alpha,\prime\beta)\ \mathit{Node})\ \mathit{VAL} \Rightarrow$
$\qquad (\prime\alpha,\prime\beta,\prime\gamma,\prime\delta)\ \mathit{list\_state} \Rightarrow \mathit{oid} \Rightarrow \mathit{bool}$
$\mathit{is\_handle\_or\_oid}\ \mathit{self}\ s\ \mathit{oid} \equiv \mathit{is\_handle\_for\_Node}\ \mathit{self}\ \mathit{oid} \vee \mathit{l1\_OclOid}\ \mathit{self}\ s = \mathit{oid}$

We will not give a complete presentation of the calculus, as the amount of lemmas is too large. However, many of them are very similar and can thus be sorted into several categories.

These categories are as follows:

1. Unfolding lemmas
2. Lemmas that state that an expression is defined
3. Lemmas stating that a correct object in a state will remain a correct object in an updated state
4. Lemmas about the level 1 oid
5. Lemmas about the value of a freshly set attribute
6. Lemmas which ensure that attributes which won't get set will remain the same
7. Variations of updext_charn4 (an object doesn't change while updating another)
8. Casting Lemmas
9. Lemmas about the free memory

For each category we will give a typical example. Most proofs are omitted. Of course they have all been proved in the Isabelle theories.

Unfolding lemmas

This kind of lemmas is helpful if we have to unfold a definition. Let's look at the following example:

lemma *set_color_unfold*:
$[\![$ *Cnode_in_state* σ (*l1_OclOid self* σ)$]\!] \Longrightarrow$
$\ulcorner$*l1_Cnode_set_color self up_color* $\sigma\urcorner$ =
updext_Cnode_in_state (*l1_OclOid self* σ)
(*base* (*Setcolor* (*access_Cnode_in_state* σ (*l1_OclOid self* σ)) (*up_color* σ))) σ

These lemmas do a little more than just unfolding the definition. They perform some additional simplifying for "normal" cases, e.g. if the object is indeed a `Cnode` in the state in this example. Their proofs are very simple, usually an unfolding of the definition and a simplifier, invoked with several additional lemmas. They are most often to be used with the *subst* command. As they usually contain some assumptions, the simplifier alone would not be enough. Additionally, the *subst* command allows to unfold only one specific occurrence of the definition in the subgoal - and not all of them.

Lemmas that state that an expression is defined

We have already stated the importance of definedness in OCL. It is also of big importance during reasoning over IMP++. During a Hoare proof, we will usually have to assure that a specific operation, state, or object will be defined. This is because many things might go wrong in an object-oriented program. For example, if we try updating an undefined object. HOL-OCL provides a large library for reasoning over definedness, what we additionally need are lemmas about the definedness of our programming language constructs.

A trivial, but helpful lemma states that an object we get from the universe is always defined. Its proof consists of one simplifier step.

lemma *DEF_get_Cnode2*:
 DEF (*level0.list.get_Cnode obj*)
by (*simp add*: *level0.list.get_Cnode_def*)

The next sort of definedness lemmas are for the setters. Of course we cannot conclude that a state which is reached after execution of a setter will always be defined, this is only the case if the object to be updated is correct and in the state.

lemma *set_n1_is_DEF2*:
 StackObject_in_state s (*l1_OclOid so s*) $\Longrightarrow$
  DEF (*l1_StackObject_set_n1 so foo s*)
by (*simp add*: *l1_StackObject_set_n1_def Let_def*)

Finally we need lemmas ensuring that a specific level 1 object is defined.

lemma *DEF_self_s_Node*:
 ⟦*is_handle_for_Node self oid*; *Node_in_state* σ *oid*⟧ $\Longrightarrow$
  DEF (*self* σ)

Lemmas stating that a correct object in a state will remain a correct object in an updated state

Most lemmas and definitions have as assumption a variation of *object_in_state* σ *oid*. We therefore need some lemmas for the reasoning over these expressions. They assure that they get propagated over a correct update. There are a lot of possible combinations. In our example we have three classes and six setters, which already makes eighteen. Additionally there are also lemmas necessary for state transitions through a create and update_new.
Fortunately, most of these lemmas are rather easy to prove: unfold the definition such that the proof goal is rewritten to an update operation, and then apply the corresponding rule for the update.

lemma *Node_in_state_set_size*:
 ⟦*Node_in_state* σ *oid*; *l1_OclOid self* σ = *oid*⟧ $\Longrightarrow$
  Node_in_state ⌜*l1_Node_set_size self foo* σ⌝ *oid*
   apply (*simp add*: *l1_Node_set_size_def Let_def*)
   apply (*rule Node_in_state_update*, *assumption*)
done

There is a notable variation of this kind of lemmas. They state that an object which will be created during an update_new will be a correct member of the state. The proof of these lemmas is a little more complicated, however only one such lemma is necessary for each update_new definition.

An example of such a lemma is the following:

lemma *becomes_ Cnode_ in_ state_ new_ n1*:
$[\![$*is_ handle_ for_ StackObject self oid*; *StackObject_ in_ state* σ *oid*; *memory_ not_ full* σ $]\!] \Longrightarrow$
Cnode_ in_ state $\ulcorner$*l1_ StackObject_ update_ n1_ new self* $\sigma\urcorner$
(*level0.list.StackObject.n1* (*self* $\ulcorner$*l1_ StackObject_ update_ n1_ new self* $\sigma\urcorner$))

Variations of updext_charn4

This category is quite important. We have to assure that the value of an object *self* will be the same if interpreted in some state σ and σ', if σ' was created through a correct update of another object in state σ. The following lemma is a typical member of this sort of lemmas: if we update any `Cnode`, our stack object will stay the same:

lemma *so_ stays_ Cnode_ updates*:
$[\![$*is_ handle_ for_ StackObject so so_ oid*; *StackObject_ in_ state* σ *so_ oid*;
Cnode_ in_ state σ *oid*$]\!] \Longrightarrow$
so σ = *so* (*updext_ Cnode_ in_ state oid foo* σ)

From this lemma we can then derive more specific ones:

lemma *so_ stays_ set_ color*:
$[\![$*is_ handle_ for_ StackObject so so_ oid*; *StackObject_ in_ state* σ *so_ oid*;
Cnode_ in_ state σ (*l1_ OclOid self* σ)$]\!] \Longrightarrow$
so σ = *so* $\ulcorner$*l1_ Cnode_ set_ color self foo* $\sigma\urcorner$

Lemmas about the level 1 oid

We have already explained why we can not have a handle for the local objects. We therefore have to get the oid of such a level 1 object through other means.

First, we provide a handful of lemmas which ensure the correspondence between the level 0 value of the attributes of the stack object, an oid, with the level 1 oid value of the corresponding object.

lemma *l1_ oid_ is_ so_ n2*:
$[\![$*Node_ in_ state* σ (*level0.list.StackObject.n2* (*so* σ))$]\!] \Longrightarrow$
l1_ OclOid (*l1_ StackObject_ n2 so*) σ = *level0.list.StackObject.n2* (*so* σ)

Together with some unfolding, simplification and reasoning about definedness, the proof of this lemma requires the following lemma:

lemma *oidOfX_ Node*:
$[\![$*Node_ in_ state* σ *self_ oid*; *access_ Node_ in_ state* σ *self_ oid* = *x*$]\!] \Longrightarrow$
OclOidOf0 x = *self_ oid*

The second kind of lemmas in this category state, that the oid will not change during an update. They are basically variations of the next category.

Lemmas which ensure that attributes which won't get set will remain the same

The just mentioned lemmas about the oid are the most important ones of this category. We have however added also several others, for example one stating that the `color` of the *N1* object will stay the same while setting the one of the *N2* object. If the object whose attribute should remain is different from the one we are updating, the proof is easy: basically an application of one of the lemmas which ensure that an object does not change during another update. It is therefore not necessary to include lemmas for each of these cases. More effort is needed if we talk about the same object, in that case it's inevitable to unfold the setting definition even on level 0. The following lemma is an example of such a theorem. Here, matters are complicated even further through the fact that the two attributes are members of different classes.

lemma set_next_color_stays_n1_n1:
$[\![$is_handle_for_StackObject so so_oid; StackObject_in_state σ so_oid;
Cnode_in_state σ (l1_OclOid (so .n1) σ); Node_in_state σ (l1_OclOid (so .n2) σ);
l1_OclOid (so .n1) $\sigma \neq$ l1_OclOid (so .n2) σ; foo σ = (so .cn1 .color) $\sigma]\!] \Longrightarrow$
(so .cn1 .color) $\ulcorner$l1_Node_set_next (so .n1) (so .n2) $\sigma\urcorner$ = foo σ

Lemmas about the value of a freshly set attribute

Of course we need to know what the value of a freshly set attribute is, as in the following example.

lemma color_of_set_color_n1:
$[\![$Cnode_in_state σ (l1_OclOid (so .n1) σ); is_handle_for_StackObject so so_oid;
StackObject_in_state σ so_oid$]\!] \Longrightarrow$
l1_Cnode_color ((so .cn1)) $\ulcorner$l1_Cnode_set_color (l1_Node_2_Cnode (so .n1)) foo $\sigma\urcorner$ = foo σ

Lemmas about the free memory

Most of these lemmas have already been introduced earlier. What is needed additionally is that the free memory will decrease by at most one during an update_new.

lemma enough_space_update_new_n1:
$[\![k > 0$; enough_space_for σ k; StackObject_in_state σ oid;
is_handle_for_StackObject self oid$]\!] \Longrightarrow$
enough_space_for ($\ulcorner$l1_StackObject_update_n1_new self $\sigma\urcorner$) (k−(1::nat))

An update_new consists of two state transitions. Therefore the proof of these lemmas has to combine the fact that the free space decreases by at most one element during a create, and that it remains the same during an update.

Casting Lemmas

In any program supporting inheritance, frequent type casts are common. It is therefore naturally that several lemmas are needed for this process. The basic facts of casting are already provided by the HOL-OCL library. An example of one which is still necessary is the following, stating that the oid of the level 1 object remains the same if we cast it.

lemma *l1_oid_c*:
$[\![$ *Cnode_in_state* σ (*l1_OclOid* (*so .n1*) σ); *is_handle_for_StackObject so so_oid*;
StackObject_in_state σ *so_oid* $]\!] \Longrightarrow$
(*l1_OclOid* (*so .cn1*) σ) = (*l1_OclOid* (*so .n1*) σ)

Simplifier set

Most lemmas of the calculus can safely be added to the simplifier. This way, large parts of the final Hoare proof will be conducted automatically. Namely, all the lemmas of category 2 and 3 are added.

5.2 The Correctness Proof

We are now ready to conduct the final Hoare proof over our program. We want to show that our program, called *generate_cyclic_list*, conforms to the invariant. We will show almost the full proof, providing explanations of the single proof steps. This is how the lemma looks like:

lemma *cyclic_List_hoare*:
$\models$ {λ σ. $\neg$($\sigma \vDash$ *err*) $\wedge$
    is_handle_for_StackObject so so_oid $\wedge$
    StackObject_in_state $\ulcorner\sigma\urcorner$ *so_oid* $\wedge$
    enough_space_for $\ulcorner\sigma\urcorner$ *2*}
generate_cyclic_List so
{λ σ. $\neg$($\sigma \vDash$ *err*) $\wedge$
Cnode_inv ((*so .return*) $\ulcorner\sigma\urcorner$) $\ulcorner\sigma\urcorner$}

If the program, invoked on the StackObject *so*, starts in a non-error state, where *so* is defined and has the handle *so_oid*, and where the memory can at least store two more objects, then we know that the state which is reached after the execution is not the error-state, and the returned object satisfies the invariant.

We start the proof with unfolding the definition of the program.

apply (*simp add*: *generate_cyclic_List_def*)

There are two main possibilities to conduct this proof. One is to start applying a sequence of applications of the rules *semi_hoare*, *conseq_hoare* and the Hoare rules for the specific commands, and start from the last command of the program and go backwards. Here we opted for the second possibility of starting with the first command and giving explicit instantiations of the *semi_hoare* rule for each command.

The substitutions always follow the same pattern. After each command, we have to assure that most results we already have will also hold in the state after executing the command, plus one or two additional ones.

The first command is *so .n1* := *New*(*Cnode*)

apply (*rule_tac Q* = λ σ.
    is_handle_for_StackObject so so_oid $\wedge$

StackObject_in_state $\ulcorner\sigma\urcorner$ *so_oid* $\wedge$
enough_space_for $\ulcorner\sigma\urcorner$ *1* $\wedge$
Cnode_in_state $\ulcorner\sigma\urcorner$ (*l1_OclOid* (*so .n1*) $\ulcorner\sigma\urcorner$) **in** *semi_hoare*)

Additionally to what gets propagated, we know that the newly created object must be a `Cnode` member of the state and that there is still one slot left in the memory.

apply (*simp add*: *hoare_ss*)
apply *clarify*

With the above two applications we unfold the Hoare triple and simplify the subgoal. The simplifier set *hoare_ss* contains the lemmas *hoare_valid_def*, *localValidDefined2sem*, *C_cmd* and *err2sem*.

With the command *subgoal_tac*, we add to the assumption list the fact that our memory is not full. This fact is used by many lemmas in the simplifier which reason about the create operation.

apply (*subgoal_tac memory_not_full* $\ulcorner s\urcorner$)

The simplifier is able to proof all but the two subgoals about the memory automatically.

apply *simp*

Next we prove the first fact about the memory:
enough_space_for $\ulcorner$*l1_StackObject_update_n1_new so* $\ulcorner s\urcorner\urcorner$ (*Suc 0*)

apply (*rule_tac t* = (*Suc 0*) **and** *s* = (*2*: : *nat*) $-$ *1* **in** *subst*, *simp*)
apply (*rule enough_space_update_new_n1*, *simp_all*)

And finally the subgoal that we have added.

apply (*erule enough_space_implies_not_full*, *simp*)

We can go on to the next command: *so .n1* := *New*(*Cnode*).
We have to propagate all facts from before except the one about the memory, as we will not create any new object anymore. Additionally, we know the fact about our new `Cnode` and that it is not equal to the one we created before.

apply (*rule_tac Q* = λ σ.
is_handle_for_StackObject so so_oid $\wedge$
StackObject_in_state $\ulcorner\sigma\urcorner$ *so_oid* $\wedge$
Cnode_in_state $\ulcorner\sigma\urcorner$ (*l1_OclOid* (*so .n1*) $\ulcorner\sigma\urcorner$) $\wedge$
Cnode_in_state $\ulcorner\sigma\urcorner$ (*l1_OclOid* (*so .n2*) $\ulcorner\sigma\urcorner$) $\wedge$
(*l1_OclOid* (*so .n1*) $\ulcorner\sigma\urcorner$) $\neq$ (*l1_OclOid* (*so .n2*) $\ulcorner\sigma\urcorner$) **in** *semi_hoare*)

The beginning is as for the previous command.

apply (*simp add*: *hoare_ss*)
apply *clarify*
apply (*subgoal_tac memory_not_full* $\ulcorner s\urcorner$)
apply *simp*

To prove that the new oid is not the same as the old one, we have a lemma available.

apply (*rule not_sym*, *rule new_n2_oid_not_old*, *simp_all*, *erule Cnode_in_dom*)
apply (*rule enough_space_implies_not_full*, *assumption*, *simp*)

The next command (*so .cn1 .color* := T) is fairly standard: We have to propagate everything we had before plus prove the fact that the color of our *N1* object is set to true.

apply (*rule_tac Q* = $(\lambda\ \sigma.\ ?P\ \sigma \wedge ((((so\ .cn1)\ .color)\ \ulcorner\sigma\urcorner) = (\mathsf{T}\ \ulcorner\sigma\urcorner)))$ **in** *semi_hoare*)

In this rule application, *?P* is denoting the current pre-assertion.

apply (*simp add*: *hoare_ss DEF_ss*)
apply *clarify*
apply *simp*

The following step is needed for proving that T is the same in every state. *lifting_ss* is the simplifier set for the lifting operators of HOL-OCL, *OclTrue_def* the definition for T.

apply (*simp add*: *lifting_ss OclTrue_def*)

The command (*so .cn1 .color* := F) is completely analogous to the one before.

apply (*rule_tac Q* = $\lambda\ \sigma.\ ?P\ \sigma \wedge$
$(((so\ .cn2)\ .color)\ \ulcorner\sigma\urcorner) = (\mathsf{F}\ \ulcorner\sigma\urcorner)$ **in** *semi_hoare*)
apply (*simp add*: *hoare_ss DEF_ss*)
apply *clarify*
apply *simp*
apply (*simp add*: *OclFalse_def OclTrue_def lifting_ss*)

After the command (*so .n1 .next* := (*so .n2*)), we additionally know that the next attribute of n1 points to n2.

apply (*rule_tac Q* = $\lambda\ \sigma.\ ?P\ \sigma \wedge$
$(((so\ .n1)\ .next)\ \ulcorner\sigma\urcorner) = (so\ .n2)\ \ulcorner\sigma\urcorner$ **in** *semi_hoare*)
apply (*simp add*: *hoare_ss*)
apply *clarify*
apply *simp*
apply (*simp add*: *OclFalse_def OclTrue_def lifting_ss*)
apply (*rule next_of_set_next_n1*, *simp_all*, *erule not_sym*)

Analogously for the command *so .n2 .next* := (*so .n1*):

apply (*rule_tac Q* = $\lambda\ \sigma.\ ?P\ \sigma \wedge$
$(((so\ .n2)\ .next)\ \ulcorner\sigma\urcorner) = (so\ .n1)\ \ulcorner\sigma\urcorner$ **in** *semi_hoare*)
apply (*simp add*: *hoare_ss*)
apply *clarify*
apply (*simp add*: *OclFalse_def OclTrue_def lifting_ss*)
apply (*rule next_of_set_next_n2*, *simp_all*, *erule not_sym*)

And finally the last command: *so .return* := *so .n1*. We first apply the standard Hoare rules and show that the final state is defined.

apply (*simp add*: *hoare_ss*)

apply *clarify*
apply (*rule conjI*)
apply *simp*

We have arrived at the crucial part of our lemma. From all the facts we gathered so far through the program, we have to show that our invariant is satisfied. This is how the subgoal looks like at this point:

Cnode_ inv ((*so .return*) ⌜*l1_ StackObject_ set_ return so* (*so .n1*) ⌜*s*⌝⌝) ⌜*l1_ StackObject_ set_ return so* (*so .n1*) ⌜*s*⌝⌝

We first have to unfold the definition of our invariant:

apply (*simp add*: *Cnode_ inv_ def Cnode_ inv_ enc_ def*)

Then apply a rule about the image of a function.

apply (*rule image_ eqI*)
apply (*rule refl*)

We now have to tackle the greatest fixpoint using the weak coinduction rule with an explicit instantiation: the only two `Cnodes` we have in our program.

apply (*rule_ tac X* = {*so .n1*, *so .n2*} **in** *weak_ coinduct*)
apply *simp*
apply *simp*

The current subgoal is a conjunction of validity statements in the final state, plus two inequalities: in this final state, the `color` of *N1* must not be the same as the one of its `next` attribute, and analogous for *N2*.
With the next lemma, the validity statements are transformed to standard definedness statements.

apply (*simp add*: *localValidDefined2sem*)
apply (*rule conjI*)
apply (*simp add*: *OclFalse_ def OclTrue_ def lifting_ ss*)
apply (*rule conjI*)

Now there remain the two inequality statements. We omit the rest of the proof. It starts with unfolding the inequality operator which is the strict OCL version of the traditional inequality operator, so additional definedness steps have to be performed. The rest is then fairly standard: we know the values of the `next` and of the `color` attributes. There are however still some steps necessary, as there are many type casts between `Node` and `Cnode` objects.

With proving this lemma, we have shown that the implementation of the method guarantees that the returned object fulfils the invariant when started in a defined state. Hereby we can state that the method implementation conforms to its specification.

6 Conclusion

In this thesis we have shown the feasibility of a program verification approach for object-oriented software. Based on an existing embedding of an imperative language in Isabelle/HOL, we provided a formal semantics to a small object-oriented language called IMP++. Using a Hoare logic, we are able to conduct correctness proofs over a program in IMP++ and establish that an implementation conforms to a specification.

This work was based on the existing HOL/OCL framework. This has two big advantages. At first, we are able to reuse large parts of the library of HOL/OCL and inherit its advantages. Particularly, it is a nice feature that both the semantics over a specification and over an implementation can use the same store model. Also the provided reasoning over undefinedness and three-valuedness turns out to be very useful for reasoning over an object-oriented programming language semantics.

Equally important is the fact that through extending HOL-OCL we now have one single, integrated approach. We are able to specify our model in UML and formalise the big step semantics in OCL. Using the same object store, we can define the operations in IMP++ and reason about the program using small-step semantics, having a clear correspondence to the specification. We thus have one integrated code verification method for IMP++ programs with a UML/OCL specification.

For being able to reason efficiently over a program, we established a large calculus for reasoning over state transitions. It would not be feasible to have to prove them for each model from scratch again. However, this will not be necessary as they will finally all be established by the encoder while loading the model. We believe that only the final Hoare proof will require user assistance. For all definitions and the lemmas, we tried to employ a modular structure which was already heavily used throughout the HOL/OCL library. This way we ensure the scalability of our approach.

Currently the extension of the encoder to create this calculus is not finalised yet. However, we were developing the calculus in a way which makes the automation quite easy, so it can be expected that this will be done in the near future.

The methodology is derived completely conservatively. The HOL-OCL framework also uses only conservative extensions. We can therefore guarantee by construction the consistency of the formalisation.

Although IMP++ is not a full-blown object-oriented programming language yet, the supported subset is quite large. We were able to support all important concepts except method calls using a very tiny datatype definition. Many concepts which are traditionally modelled separately, e.g. getters, setters, type casts, or assignments, can be modelled with a simple transition on states. Of course, this puts the crucial parts into the formalisation of the state. And actually the state invariant is of big importance for our calculus.

Large parts of the verification can be automatised. The library will in the end be created by the encoder and also the final Hoare proof as presented in Section 5.2 can be further automatised with introducing specific tactics. However, we believe that a Hoare logic will only be an intermediate step. Hoare logic is not well suited for efficient verification of large programs. But it provides the basis for enhanced verification techniques like a weakest precondition calculus and a verification condition generator. This could lead to effective program verification techniques based entirely on derived rules.

So far, HOL-OCL is geared towards refinement, potentially supported by automated model-transformations, and code-generation from suitably refined models. The main drawback was so far its lack of amenability to legacy or library code. In the end, HOL-OCL should support the full software development process, such that critical parts of a system can be developped in a fully formal way [3]. A full, formal tool-chain should also include automatic test-case generation and code-generation. With our work, we enhance the tool chain with a post-hoc verification method, and widen the applicability of HOL-OCL.

6.1 Related Work

There is a substantial body of literature about the topic of object-oriented language semantics. There are several projects which aim to embed a subset of Java into higher-order logic, among them NanoJava [11] or the Isabelle/Bali [10] project. These works use a deep embedding of the language. Syntax, semantics and types are represented by free datatypes. As as consequence, the calculi has a heavy syntactic bias in form of side-conditions over binding and typing issues. While this is unavoidable if one is interested in meta-theoretic properties, this is a major obstacle for efficient deduction. Therefore, none of these deep embeddings has been used for substantial proof work in applications.

The KeY tool is an integrated formal specification and verification environment for specifications consisting of Hoare-style annotations of Java programs. It offers remarkable support for development and has a high degree of proof automation. However, it is built by definitional axioms and thus has not formally investigated the issue of consistency. Moreover it doesn't fully comply to the OCL standard with only supporting a two-valued logic.

Jive compiles an object-oriented data-model into a fixed Isabelle theory and includes this into a derived Hoare-calculus over a substantial fragment of Java. As HOL-OCL, the construction is done with a shallow embedding, however, it is based on a closed-world assumption and thus not extensible.

There are also some well developed verification condition generator approaches. One example is Boogie for Spec#. The underlying idea is to compile object-oriented programs into standard imperative ones and to apply a verification condition generator on the latter. The approach requires the generation of a quite substantial axiomatisation of an object-oriented memory model, and an explicit first-order representation of object-oriented types within a logical context in which the verification conditions were stated.

6.2 Future Work

While this thesis provides the basis and shows the feasibility of a typed object-oriented verification approach, there are several possibilities for future research. The most important points are:

- Extend the encoder so it automatically creates the vast majority of definitions and lemmas.
- Provide support for method calls and invocations.
- Derive a verification condition generator.

To extend the encoder would probably be the next step to take. The method could then be tested against a larger library of examples. The library of lemmas which have to be generated is very large. However, we developed the library in a very modular and generic way, which makes the encoding easy.

Even tough the syntax of IMP++ is quite small, most object-oriented features can be integrated. But it still has to be extended to count as a full-blown object-oriented programming language. The most important feature which is currently missing are the method calls. This would probably require quite substantial work.

The calculus for IMP [5] uses Hoare logic for a weakest precondition calculus and derives a verification condition generator. This should be implemented for IMP++ as well. The automation of a Hoare proof can thereby be improved drastically, making even verification of large programs feasible.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master thesis for Lukas Brügger

Proof Support for IMP++ in HOL-OCL

Supervisors:	Achim Brucker and Burkhart Wolff
Professor:	Prof. D. Basin
Issue Date:	January, 30 2007
Submission Date:	July, 29 2007

Introduction and Motivation

HOL-OCL [1] (http://www.brucker.ch/projects/hol-ocl/) is an interactive proof environment for UML/OCL. Its mission is to give the term "object-oriented specification" a formal semantic foundation and to provide effective means to formally reason over object-oriented models. On the theoretical side, this is achieved by representing UML/OCL as a conservative, shallow embedding into the HOL instance of the interactive theorem prover Isabelle [3]. We follow the standard [4] as closely as possible, in particular, we prove that inheritance can be represented inside the typed λ-calculus with parametric polymorphism. As a consequence of conservativity with respect to HOL, we can guarantee the consistency of the semantic model. On the technical side, this is achieved by automated support for typed, extensible UML data models.

At present, with respect to a system development method, HOL-OCL is geared towards refinement (potentially supported by automated model-transformations) and code-generation from suitably refined models. The main drawback of this approach is its lack of amenability to legacy or library code. A post-hoc verification method may complement the model transformation approach and widen its applicability.

Assignment

Objective

The objective of this master thesis, following the lines of [2], is to improve tool-support for a code-verification method based on Hoare-calculus. This code-verification methods should support a a small object-oriented programming language IMP++. While the Hoare-rules for the main IMP++ constructs like assignment, conditional, and while-loop have already been derived from a conservative definition of a denotational semantics for IMP++, the part of the language related to object-creation/destruction and method-calls (and method-invocations) is still generic and needs to be specialized to each given data-model. Here, it is the goal to develop specific tactics and integrate them into the data-type package inside HOL-OCL.

Tasks

The following tasks are mandatory:

- build up a cyclic object structure via a (hand-crafted) theory of constructors and destructors and prove some simple properties in several well-chosen examples,
- build up a cyclic object structure via a (hand-crafted) theory of calls or invocations and prove some simple properties in several well-chosen examples using method calls or invocations,
- mechanize the generation of the constructor and destructor theories, and
- mechanize the generation of the theory for calls or invokes.

The following tasks are optional extensions:

- derive a wp-calculus and an awp operator for IMP++ (following the IMP example) and derive a verification condition generator (VC),
- build a (hand-crafted) data-type theory for VC and verify a small program using VC,
- build a (hand-crafted) method call or method invocation theory and verify a small program using VC, and
- mechanize the hand-crafted parts.

Deliverables

- At the beginning of the thesis, an agreement must be signed which allows the supervisors of this thesis, his project partners, and ETH Zurich to use and distribute the software written during the thesis.
- At the end of the first week of the thesis, a time schedule of the master thesis must be given and discussed with the supervisors. Regular meetings are expected to be hold between the supervisor(s) and the student.
- At the end of the master thesis, a presentation of 30 minutes must be given during an Infsec group seminar. It should give an overview as well as the most important details of the work.
- The final report may be written in English or German. It must contain a summary written in both English and German, this assignment and the schedule. It should include an introduction, an overview of related work, and a detailed description of the software implemented. Four copies of the final report must be delivered to the supervisor.
- Software and configuration scripts developed during the thesis must be delivered to the supervisor on a CD-ROM.

References

[1] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zürich, 2006.

[2] A. D. Brucker and B. Wolff. A package for extensible object-oriented data models with an application to IMP++. In A. Roychoudhury and Z. Yang, editors, *International Workshop on Software Verification and Validation (SVV 2006)*, Computing Research Repository (CoRR). Seattle, USA, Aug. 2006.

[3] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002.

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

Unterschrift: ....

Bibliography

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

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

[3] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. An MDA framework supporting OCL. *Electronic Communications of the EASST*, 5, 2006.

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

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

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

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

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

[9] Dave Power, Bertrand Meyer, Jack Grimes, Mike Potel, Ron Vetter, Phil Laplante, Wolfgang Pree, Gustav Pomberger, Mark D. Hill, James R. Larus, David A. Wood, Hersham El-Rewini, and Bruce W. Weide. Where is software headed? a virtual roundtable. *Computer*, 28(8):20–32, 1995.

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

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

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