Verification Support for Object Database Design

David Spelt

Samenstelling van de promotiecommissie:

Prof. dr. P.M.G. Apers, promotor
Prof. dr. G. Weikum, University of the Saarland, Saarbruecken, Germany
Prof. dr. H.C.M. de Swart, Tilburg University
Prof. dr. ir. E. Brinksma
Prof. dr. ir. W. Jonker
Dr. H. Balsters, assistent-promotor
Dr. B. Jacobs, University of Nijmegen, referent

School voor Informatie- en KennisSystemen (SIKS)
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
SIKS Dissertation Series No. 99-7

Centre for Telematics and Information Technology (CTIT)
P.O. Box 217, 7500 AE Enschede, The Netherlands
CTIT Ph.D-thesis Series No. 99-24

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

ISBN: 90-365-1355-3
ISSN: 1381-3617; no 99-24 (CTIT Ph.D-thesis series)
Printed by: Printpartners Ipskamp, Enschede, The Netherlands

Copyright ©1999 by David Spelt, Enschede, The Netherlands

VERIFICATION SUPPORT FOR OBJECT DATABASE DESIGN

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. F. A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 10 september 1999 te 15.00 uur.

door

David Spelt

geboren op 29 december 1970
te Papendrecht

Dit proefschrift is goedgekeurd door:

Prof. dr. P. M. G. Apers (promotor)
Dr. H. Balsters (assistent-promotor)

Chapter 1

Introduction

1.1 Information system design

Information system technologies have become an integral part of people's daily lives and businesses. Many daily-life activities, such as the taking of cash from an ATM machine or the use of a cellular phone, involve the application of complex information systems: e.g., the use of a cellular phone involves the consultation of complex routing and billing systems behind the scenes. With the increasing dependence of society on information systems, their correct functioning becomes an important issue.

Early 1999, the US Presidential Information Task Force Advisory Council (PITAC) submitted a report ([JK99]) to President Clinton containing recommendations on future US government funding priorities in Computer Science. The report cited several recommendations aimed at improving the quality of software. One of the findings in the report is that current technologies for information system engineering are inadequate to build reliable software: "there is virtually no ability to perform accurate engineering analysis, little re-use of previously developed components, and no way to know the extent to which a large software system has been tested". Furthermore, it is argued that "the construction and availability of libraries of certifiable robust, specified, modeled, and tested software components would greatly aid the development of new software".

Component based software development through the use of so-called object-oriented programming languages is emerging. Being able to validate the correctness of large and complex software components is a challenging goal of Computer Science research.

In this thesis, we present a formal method to validate the correctness of object-oriented programs. Particularly, our focus is on developing tool support for the validation of database programs. Being able to validate the correctness of database programs is of major importance, because databases often play a central role in the information system infrastructure of an organization. Correctness of the data that is stored is crucial for the correct functioning of the various application programs that make use of this data. This makes it worthwhile to invest in the effort of formal verification.

1.1.1 Validation, testing and verification

The development of reliable software suggests that some kind of validation support is needed. An important first step to make such validation feasible is to give a precise definition of the intended behavior of the system. A precise (formal) description of the task to be performed by a system is called a *specification*: it asserts *what* the system is supposed to do, rather than *how* this is done. A specification deals with a number of *assertions* (i.e., logical statements that should always be true for the system—see [Mey97]). Validation of the correctness of an *implementation* of the system, i.e. a real executable computer program, now amounts to checking its compliance to the specified assertions. Two different approaches can be distinguished towards system validation. These approaches are known as testing and verification [Mey85].

Testing is a method to validate the correctness of an implementation by experimentation. The correctness of a program is tested by examining its behavior for a selected set of input data. Simple debugging of programs by *ad-hoc* ("trial-and-error") experimentation with several different input data is certainly the most widely used method for the validation of software. Formal (conformance) testing yields a more systematic approach, in which appropriate test cases are derived from a specification. Errors may be discovered in a design by examining the output of the implementation for the selected test data. In practice, however, testing is often incomplete because only a small subset of the possible input data can be examined within a reasonable amount of time.

Verification is a method to prove the correctness of an implementation with respect to a specification by mathematical reasoning. It has the advantage over testing that it can give a high degree of certainty about the correctness of an implementation: correctness of the implementation is proved formally, once and for *all* possible input states, rather than testing for a limited number of cases. The practical use of this method in real-life software development, however, is not without difficulties. Small systems may be verified easily by hand, but this is not feasible for many realistically-sized systems: proofs tend to be large, full of intricate details, which makes the verification difficult to manage if done by hand. Furthermore, fully automatic verification methods are inherently limited in scope; the assertions we would like to verify are often undecidable.

Despite of the shortcomings of the aforementioned approaches, both testing and verification are feasible choices for practical system validation. In this thesis, our focus is on verification, not on testing.

1.1.2 Approaches to verification

There are two well-established approaches to verification, namely *model checking* and *theorem proving*[EMC96]. *Model checking* is a technique for checking an assertion about a system by examining all possible states. Using a finite model of the system, an exhaustive state space search is performed to check that the desired property holds. A main advantage of model checking is that it produces counterexamples. This provides a valuable form of feedback to designers for debugging a system ([EMC96]). Model checking is complicated, however, by the so-called state explosion problem. Model checkers today can handle systems with up to a few hundred state variables ([EMC96]). Using appropriate abstraction techniques, a larger number (even an infinite number) of states may be within the reach of a model checker.

Theorem proving is a technique to prove the correctness of a system by formal reasoning using logic. To verify the correctness of a computer program with respect to a specification, one first has to assign a meaning to that program. The meaning of a programming language is known as its *semantics*, usually *denotational semantics* [Win93]. Denotational semantics provides a mathematical (functional) description of the input/output behavior of a program. Using this description, the behavior of a program can be reasoned about formally: proving an assertion about the program's behavior amounts to verifying a *theorem* in logic.

Different logical formalisms can be used to reason about the behavior of a computer program, e.g. first-order logic, higher-order logic, or dynamic logic—each with its own specific applications. Practical verification is often carried out using higher-order logic (HOL). This formalism has the advantage of being extensively supported by tools, so called *theorem provers*. Combined with a denotational semantics approach, a theorem prover can be used to implement a reliable program verification method.

Theorem proving by machine has greatly advanced over the past decade, using higher-order logic based theorem provers such as Isabelle [Pau94, Isa] and PVS [ORR^{+}96]. Both tools enable a combination of interactive and automated verification. A main challenge in practical program verification work is to attempt to automate the verification as much as possible. The cases that cannot be proved automatically may then be examined further using the interactive mode of the theorem prover. An important topic of this thesis is to study the use of the automated mechanics of a particular theorem prover, namely the Isabelle prover, to assist in verifying several assertions about an object-oriented database design.

Both model checking and theorem proving are potentially feasible approaches to verifying databases. In this thesis, we confine ourselves to studying the theorem proving approach. Model checking has been used primarily in hardware and protocol verification; a recent trend, however, is to apply this technique to analyzing specifications of software systems ([EMC96]). This approach could also prove valuable in the context of databases, although the state explosion problem also manifests iteself here: databases typically have an infinite number of states.

1.1.3 Database requirement specification

Traditionally, a primary task of the database system is to ensure the *consistency* of the data being stored. To that end, designers specify assertions about the integrity of the data (such as key constraints), e.g.:

"*All persons in the personnel database should have a unique social security number.*"

Programs that update the database must preserve these rules, called *integrity constraints* or *invariants*: assuming that the constraint holds prior to the update, it should be the case that the constraint still holds afterwards.

Some database systems support constraint management, where the correctness of constraints is monitored at run-time by the database management system. Ideally, run-time checking of constraints would not be necessary, if it could be proved at compile-time that update operations do not violate constraints.

In the past decade, so-called *semantics-based transaction models* have emerged to meet the demands of challenging new applications of database technology such as workflow management and cooperative work. The use of these models gives rise to several additional assertions about the operations (other than invariants). For example, various models rely on *compensating transactions* , where for each transaction T an “inverse” operation T^{-1} has to be defined; T^{-1} is used internally by the transaction manager to compensate the effects of an invocation of T. Although the correctness of the model relies on the correctness of the implementation of the compensating transactions, it is often only assumed (implicitly) that these implementations are indeed correct.

Another semantics-based transaction management idea is the notion of *commutativity*. Informally, commutativity of two operations means that the operations do not conflict with each other. The idea is that by identifying cases where the operations do not conflict, the system can offer more concurrency. It is often assumed that the programmer specifies, for each operation pair, a precondition for when these operations commute. To ensure reliable operation of the database, these specifications should, however, be validated against the implementation code.

A broadly adopted approach in requirement specification and data modeling nowadays leans on object-oriented techniques. Object-oriented techniques provide high-level modeling capabilities which are declarative in nature, thus making them suitable for verification purposes.

1.1.4 Object-oriented database programming

The use of object-oriented techniques has been accepted in a broad range of information system development. Visual object-oriented modeling techniques (such as UML) are used in the initial stages of information system design, for data modeling purposes. For application programming, Java has quickly become one of the most popular (object-oriented) programming languages. Object-orientation is also emerging in the field of databases. Object-oriented data models provide rich data structuring capabilities, which provide definite advantages over the flat table structures offered by the traditional relational data model. Relational database vendors have recognized this need for more advanced data models and now try to enhance their existing products with object-oriented language features (the so-called object-relational databases). Also complete new systems have entered the market, which provide support for objects merely by adding persistence and transaction management to an object-oriented programming language. Efforts in this direction have resulted the definition of the ODMG standard ([CB97]). Both industry and the database research community recognize the importance of this standard.

1.2 Problem statement and overview of this thesis

The topic of this thesis can now be defined as follows.

> *How can we offer database designers a tool to assist in verifying requirement specifications of an object-oriented database design, using the existing generation of higher-order logic theorem provers?*

To address this question, we distinguish two main tasks.

1. **A formal translation of the object-oriented database schema to higher-order logic** This issue is addressed in the first part of this thesis. It focuses on the definition of a translation of object-oriented database schemas to higher-order logic through a so-called *semantic embedding*.

   Chapter 2 discusses the position of this work within the vast field of research in object-oriented language and system design. We clarify the assumptions that are made in this thesis with respect to typing and semantics and discuss the features that are rather specific for databases.

   Chapter 3 defines an object-oriented database programming language, called OASIS. OASIS incorporates a relevant subset of ideas from object-oriented database programming that will be used to illustrate the verification approach.

   Chapter 4 introduces the higher-order logic (HOL) Isabelle theorem prover. It introduces the HOL notation and the basics of mechanical theorem proving in Isabelle.

   Chapter 5 defines a formal translation of OASIS database schemas to higher-order logic. An algorithm is presented for the translation of a database schema to a logical theory. The main issue here is the translation of the various object-oriented language features of the database programming language. To this end, we present a new HOL theory of objects. The definition of this theory and the semantic embedding of object-oriented database schemas in logic based on this theory provides the first major contribution of this thesis.

   Chapter 6 discusses an implementation of the OASIS/HOL embedding. The implementation provides a front-end compiler to the Isabelle theorem prover. The compiler automatically generates a HOL theory from an OASIS database schema, in a format that is understood by the theorem prover.

2. **Tool support for practical verification using a theorem prover** In this part, the focus is on applications of the semantic embedding of object-oriented database schemas in logic. Practical verification using the Isabelle theorem prover is investigated, with an emphasis on automating proofs. We discuss two applications of the tool.

   Chapter 7 exploits the OASIS/HOL embedding for the task of verifying that database updates preserve a number of specified integrity constraints. Verification support is developed using the Isabelle theorem prover.

   Chapter 8 studies applications of the OASIS/HOL embedding to support semantics-based transaction models for workflow and cooperative work. Tool support is provided to verify compensation and commutativity of update operations.

Finally, Chapter 9 presents conclusions of our work.

Chapter 2

Positioning of the work within the object-oriented paradigm

The context of this work is the vast field of research in object-oriented language and system design. An overview of the current state of the field is given in the standard text books "*A Theory of Objects*" by Abadi and Cardelli [AC96], and "*Object-oriented Software Construction*" by Bertrand Meyer [Mey97]. In this chapter we examine the various features discussed in [AC96, Mey97] and give motivation for the choice of features used in the thesis.

The chapter is organized as follows. Section 2.1 discusses typing issues in object-oriented languages. We discuss these from the perspective of the so-called *classical model* for class-based languages (see [AC96]). Section 2.2 discusses method overriding, dynamic typing and late binding. In Section 2.3 we discuss several issues related to semantics; the distinction is made between "value" and "reference" semantics (see [Mey97]). Several intricate issues are discussed that arise in the presence of reference semantics. Some object-oriented features discussed in this thesis are rather specific for databases; most important of these are persistent roots, as discussed in Section 2.4. Finally, Section 2.5 concludes with a summary of points that determine the position of this work in the broader context of other (and non-database) object-oriented programming language research.

2.1 Typing issues in object-oriented languages

2.1.1 Subclasssing and subtyping

In the classical model for class-based languages there is a strong correlation between subclassing and subtyping. An important characteristic of these languages is a principle known as *subclassing-is-subtyping* [AC96]; i.e., the subtyping relation ($\leq$) is fully determined by the user-specified subclass relation. For example, $Employee \leq Person$ if $Employee$ is a subclass of $Person$.

Some languages (partly) decouple subtyping from subclassing, and adopt the weaker principle known as *subclassing-implies-subtyping* [AC96]. *Structural subtyping* is a form of subtyping in which subtyping is defined directly based on the structure of types (see [AC96]). A well-known form of structural subtyping is *record subtyping* [Car88]. A record type σ is a subtype of τ if σ has the same components as τ and possibly more. For example, the record type $\langle a : int, b : string\rangle$ is a subtype of $\langle a : int\rangle$. Subtyping also applies to the record components.

Previous research in our group (see [BF91, BdBZ93, vK97]) has adopted several forms of structural subtyping, particularly record subtyping. In this thesis, we do not consider structural subtyping in the sense of [Car88, BF91]. The work presented here adopts the principle that *subclassing-is-subtyping* mentioned above. However, subtyping propagates through structured types such as records and collections. For example, $\langle a : Employee \rangle \leq \langle a : Person \rangle$ is true, provided that $Employee$ is a subclass of $Person$. However, we do not allow $\langle a : \langle b : Employee, c : int \rangle \rangle \leq \langle a : \langle b : Employee \rangle \rangle$, because this kind of subtyping is not compatable with the simple type system of HOL.

2.1.2 Classes, interfaces and types

In the classical class-based languages, a class definition provides a mix of a type description of objects and the implementation of methods. There are situations in which this mix of concepts is problematical, e.g. in case of *multiple inheritance* where a class may inherit two incompatible implementations of the same method from different superclasses.

A recent variation of the classical model offers the possibility to separate the type description of methods from the actual implementation code. Here, object types may be specified separately in what is commonly known as an *interface* [AC96]. Typically, a class may "*implement*" multiple interfaces, but only extend a single class (see e.g. Java [AG96]). When a class implements an interface, it means that the class has to provide implementation code for all methods specified in the interface.

Abstract classes provide a mix of a normal class and an interface. An abstract class is like a normal class, but it only specifies a partial implementation: implementations may be given for some methods, while others may be specified as *abstract methods* (i.e., methods with as yet no implementation—similar to declarations in an interface).

Class definitions often give rise to recursive data structures. For example, a class $Person$ may have an attribute $spouse$ that is also of type $Person$. Similarly, mutual recursive data structures are created when two classes refer to each other via attributes.

In this thesis, we consider a language based on a single inheritance model (we do not consider multiple inheritance). Class definitions may be mutual recursive. The language includes the possibility to define abstract classes, but we do not consider interfaces. The approach extends to interfaces, however, following the guidelines for abstract classes.

2.1.3 Subsumption

A characteristic property of subtyping is *subsumption* [AC96]: a value of a subtype can be used at any place where a value of one of its supertypes is expected.

There are several ways to incorporate subsumption in a type system (see [AC96]). An extension of a type system with subsumption can be achieved by adding the one characteristic rule for subsumption [AC96]:

if e of type σ and $\sigma \leq \tau$, then e of type τ (subsumption rule)

A direct consequence of this rule is that objects acquire multiple types, which may complicate the development of type checking algorithms (as pointed out in [AC96]). Another approach uses *minimum typing* ([AC96]), where each expression is assigned a unique type (if it has a type at all). A type system based on minimum types does not incorporate the subsumption rule mentioned above. Instead, each typing rule is made such that at any place where an expression of a supertype is expected, an expression of a subtype can be used instead.

The language used in this thesis adopts minimum typing; it does not include an explicit rule for subsumption. Minimum typing of expressions is needed in defining a translation of the object-oriented language to HOL.

2.2 Overriding, dynamic typing and late binding

A subclass may override definitions of methods that are inherited from a superclass. In this thesis, we adopt the classical rule for method overriding (see [AC96]): we require that the signature of a method does not change (this rule is also adopted in Java). More flexible rules for method overriding including *method specialization* (where the output type may be a subtype) and *self type specialization* (see [AC96] for details) are not studied in this thesis. Also, we do not consider *attribute hiding* (i.e., the possibility to override an attribute definition in a subclass) as available in Java (see [AG96]) and *attribute specialization* (where the type of an attribute may be refined in a subclass).

With the introduction of subsumption and due to the possibility of overriding, method application gives rise to what is known as *late binding* or *dynamic dispatch* [AC96]. Late binding is one of the characteristic features of object-oriented programming, which is addressed in this thesis. Crucial for late binding is that information about the true *run-time type* (or *dynamic type*) of an object is available, in addition to its *minimum type* (which only specifies a compile-time type). These issues are discussed in the later chapters.

Late binding is covered by the verification approach outlined in this thesis. A *single dispatch model* is used, in the sense that the dispatching mechanism only examines the run-time type of the receiver object; the types of the other parameters do not play a role.

2.3 Reference semantics and value semantics

The distinction between reference and value semantics (see [Mey97]) is important for object-oriented programming languages. Languages such as Java and Eiffel use a value semantics for primitive types such as booleans and integers, whereas reference semantics is used for all other types, namely object types and structured types such as arrays. An exception is Smalltalk, which employs the credo that “everything is an object”, including primitive values.

In this section we discuss several issues in object-oriented languages related to the use of reference semantics. In Section 2.3.1 we explain the notions of *dynamic aliasing* and *sharing*. In Section 2.3.2 we discuss the semantics of collection types in object-oriented databases, particularly the ODMG standard [CB97]. It is shown that the standard in fact supports two complementary forms of collection types, one with a value semantics and one with a reference semantics, thus deviating from the aforementioned standard rule that all non-primitive types have reference semantics. In Section 2.3.3, we discuss well-known problems with set-oriented updates as a consequence of sharing. In Section 2.3.4 we conclude with a discussion of the pitfalls of maintaining invariants for classes.

2.3.1 Dynamic aliasing and sharing

Some object-oriented languages (e.g., C++) expose the distinction between references and values in the syntax. In other languages, such as Java and Eiffel, the distinction between reference and value semantics is rather subtle because the underlying use of references for non-primitive types is *not* shown explicitly; operations on objects implicitly bypass the reference. The distinction reveals itself, however, in the meaning of parameter passing and assignment; for primitive types, assignment and parameter passing copy values, while for non-primitive types only references are copied. Reference semantics produces what is known as *dynamic aliasing*: the possibility for a single data item to be accessible through two different names ([Mey97]). Dynamic aliasing has as an immediate consequence the possibility of *sharing* of data (also called *object sharing* in object-oriented programming languages).

The following example illustrates the use of *value semantics* for primitive types in Java. Consider two integer values i and j and the following fragment of code:

$$i = 10; j = 20; i = j; i = 30; System.out.println(j)$$

The output of this program is the *initial* value of j (namely 20). The assignment $i = j$ copies the value of j, because i and j have primitive types. The subsequent assignment $i = 30$ sets the value of i to 30; clearly, it does not affect the value of j.

The following example illustrates *reference semantics* for object types in Java. Consider two $Point$ objects p and q and the following fragment of code:

$$p.x = 10; q.x = 15; p = q; p.x = 30; System.out.println(q.x)$$

We assume that $Point$ objects have (x, y) coordinates modeled as attribute values. The assignment $p = q$ copies the reference of q, not the associated coordinate values; both p and q now refer to the same object. In any subsequent computations, updates applied to p become visible via q and vice versa. The assignment $p.x = 30$ sets the x-coordinate of p to 30, which also affects q. The output of the above code is 30, not 15.

2.3.2 Reference semantics versus value semantics for collections

Collection (or container) types are types such as array, list, set and bag. General purpose object-oriented programming languages such as Java and C++ typically use reference semantics for these types.

The situation in object-oriented databases is far less clear. The ODMG object model definition ([CB97]) mentions two kinds of collections: collections with a reference semantics ("collection objects"–see page 20 of [CB97]) and collections with a value semantics ("collection literals"—see page 32 of [CB97]). The use of these two fundamentally different notions of collection types in one standard gives rise to considerable confusion. For example, there is only a subtle distinction in the syntax: e.g., the type $\mathsf{set}\langle\mathsf{Person}\rangle$ (without a capital) stands for a set literal—a non shareable value, while Set⟨Person⟩ (with a capital) is the type of a set object—a shareable value. Strangely, the distinction between collection literals and collection objects disappears in the ODL and OQL language definition (see [CB97]). In other words, it is unclear what semantics should be used for collection types in ODMG.

In O2C [BDK92], the object-oriented database programming of the O2 database system, collection types are assigned value semantics. The following example illustrates value semantics for collection types in O2C:

$$A = set(10); B = set(20); A = B; A \mathrel{+}= set(30); display(B)$$

The above fragment of code displays the initial value of B (namely $set(20)$). The assignment $A \mathrel{+}= set(30)$ updates the set A (the new value is the union of the old value A and $\{30\}$), but since values semantics is used this does not affect the value of ys.

Value semantics for collection types is also used in TM (see [BdBZ93] for details); collections are complex values, not objects.

In this thesis we assume a value semantics for collection types. We do not discuss reference semantics for collections, as is used in general purpose object-oriented programming languages such as Java and C++. Reference semantics can, however, be simulated by defining a class with a single collection-valued attribute.

2.3.3 Reference semantics and set-oriented updates

Set-oriented data processing plays an important role in databases. The *set-at-a-time* (rather than *one-object-at-a-time*) processing of bulk-queries and updates creates additional opportunities for optimization, such as parallelization ([LS93]). Set-oriented processing of updates in object-oriented databases, however, is known to be problematical due to the possibility of sharing (see [LS93, vK97]). In this section, we briefly survey these problems and discuss the solutions that have been presented in the literature.

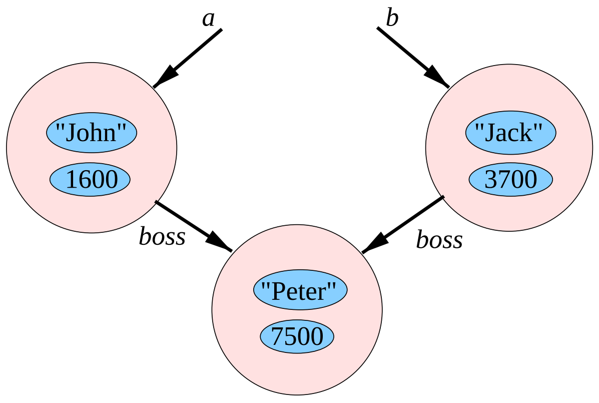

Figure 2.1: Example: object sharing

Consider the simple object configuration of Figure 2.1, which represents three $Person$ objects. We assume that $Person$ objects have three attributes—a name, a salary and a boss field (the latter also of type $Person$). John, Jack and Peter earn 1600, 3700, and 7500 pesos respectively; Peter is the boss of John and Jack. Further, the variables a and b provide handles to John and Jack respectively. In this context, the following set-oriented update is defined:

$$foreach\ x\ in\ set(a, b)\ do\ \{x.boss.sal = x.sal \times 2\}$$

The variable x iterates over the set $\{a, b\}$. For each x of type $Person$, the salary of the boss of x is set to the salary of x multiplied by 2. The outcome of this update is *non-deterministic* if a *sequential* (tuple-at-a-time) processing of the set is used: depending on the order in which the elements of the (inherently unordered!) set are processed, Peter is assigned a new salary of 3200 or 7400. Alternatively, a conflict arises if a *parallel* processing of the elements is used, where each step in the iteration observes the same input state: in this case, Peter's salary is updated with two incompatible values (namely 3200 and 7400) at the same time.

Non-determinism is an undesirable feature of a database programming language [LS93]. The parallel execution model prevents such behavior to occur, but the price to pay is that conflicts may arise due to sharing.

There are several alternative solutions to deal with conflicting parallel updates, either at run-time or at compile-time ([vK97]). In [LS93], a compile-time technique is presented to reject potentially conflicting updates. The presented technique examines the state-independent *commutativity* of the operations executed in the loop. A set-oriented update is accepted if-and-only-if the operations executed in one step of the loop commute with the operations executed in all other steps. As a simple example, consider the following update in the context of the object configuration shown in Figure 2.1:

$$foreach\ x\ in\ set(a, b)\ do\ \{x.sal = x.sal + 1\}$$

The above update is accepted because assignments of the form $x_i.a = e$ commute with assignments of the form $x_j.a = e$ for $i \neq j$; the condition $i \neq j$ is satisfied trivially because x is the iterator variable and sets do not contain duplicates.

The idea of using commutativity for identifying dependencies between operations has its background in *semantics-based concurrency control*, see [Wei88, Wei93]. We elaborate on this topic in Chapter 8.

The object-oriented database programming language discussed in this thesis includes the possibility to define set-oriented updates, with a parallel execution model underneath. The construct we use is comparable to the construct discussed in [Qia91]. We do not consider unbounded iterations and recursion with a sequential processing, as found in imperative object-oriented programming languages such as Java and C++. Furthermore, we do not elaborate on the actual detection of conflicts in set-oriented updates, although an approach for the compile-time analysis of user-specified commutativity relations for methods is sketched in Chapter 8.

2.3.4 Class invariants and reference semantics

An important topic of this thesis is the specification and static verification of invariants. The checking of class invariants in object-oriented systems is, however, known to be problemati-

```
class A export
     forward, attach
feature
     forward: B;
     attach (b1:B) is
         do
           forward := b1;
           if not b1.Void then
                 b1.attach(Current)
           end
         end
invariant
     INV : forward.Void or else (forward.backward = Current)
end

class B export
     backward, attach
feature
     backward: A;
     attach (a1:A) is
         do
           backward := a1;
         end
end
```

Figure 2.2: Example of an invariant in Eiffel

cal. As pointed out in [Mey97], problems arise due to dynamic aliasing.

Figure 2.2 shows an example of a class invariant which illustrates the problem in Eiffel. The example shows the definition of two classes A and B with a bidirectional (i.e., a forward and a backward) link between related objects. The invariant asserts that objects should correctly refer to each other. Although the method in class A preserves the invariant, problems arise because the invariant can be violated by the method of class B. This is illustrated in Figure 2.3. We assume that there are two instances a_1 and a_2 of class A and an instance b of class B. Initially the objects a_1 and b_1 are properly linked (thus satisfying the invariant). However, the result of invoking the *attach* method to b_1 with a_2 as parameter is that the *backward* link is re-routed from a_1 to a_2 thus violating the invariant. Aliasing creates the opportunity that public attributes of an object are modified by an operation on another object. Consequently, the checking of invariants cannot always be localized on the object.

In this thesis invariants are specified at the global level (and called integrity constraints), not at the class level. This excludes problems as thus described above. Applications can only modify the database contents using the updates defined in the database schema. Consistency is checked for each method with respect to the global constraints. The goal of this thesis is to develop a technique to verify at compile-time that the methods in the database schema do not

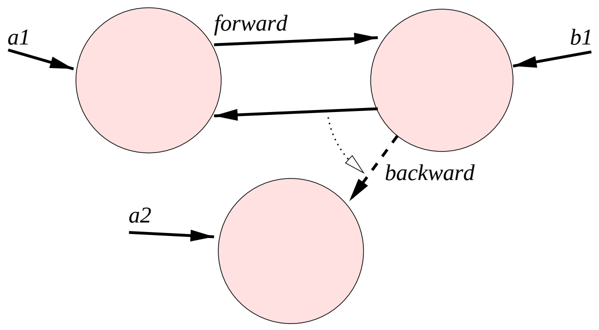

Figure 2.3: Example: violating the invariant

violate the specified constraints.

2.4 Persistent roots

In addition to the standard object-oriented language features discussed in the previous sections, a database programming language often supports features that are not available in general purpose programming languages. Obviously, the most important is persistence.

One approach to provide persistence to objects is known as *persistence by reachability*. This approach is used in the O2 [BDK92] database system. In this approach, one declares so-called *persistent roots*. These are global variables whose contents are persistent. That is, an objects becomes persistent when it is "reachable" (i.e., directly or indirectly referenced) via one of the persistent roots. An object is initially non-persistent (called *transient*). Persistence to an object is provided by updating the root variables, such that the object is made reachable. Similarly, a persistent object can become transient by breaking all references to it via the roots.

In this thesis we discuss a database programming language based on persistence by reachability. Formally, persistent roots are treated as global variables.

2.5 Summary and discussion

The language studied in this thesis can be characterized as a classical class-based language (see [AC96]). Standard class-based features covered by the approach include single inheritance, data recursion (mutual recursive class definitions are supported), method overriding, abstract classes and late binding. Value semantics is used for primitive types. Parameterized collection types (such as set⟨Person⟩) are also assigned a value semantics, in contrast to the reference semantics that is used in other general purpose object-oriented programming languages (such as Java [AG96]).

The imperative programming language that is studied in this thesis has a limited expressive power. Object creation, assignment, conditionals, sequential composition, and local variables are supported. Further, a declarative set-bounded iteration construct is available. We do not consider, however, several features of general purpose object-oriented programming languages: unbounded iteration, recursion, exceptions, and threads are not addressed in this thesis.

Chapter 3

The OASIS object-oriented database language

The previous chapter gave an overview of the important concepts in object-oriented language and system design. This chapter presents an object-oriented database language, called OASIS (which stands for "Object AnalysIS"), used for study in this thesis. The definition of this language was motivated by the research topic addressed in this thesis—verification support for object database design. The language includes a relevant subset of ideas from object-oriented database programming, which will be used to illustrate the verification approach.

The OASIS language was designed to be similar to the object-oriented database system O2 [BDK92] and the ODMG standard [CB97]. In addition, it includes facilities for global integrity constraint specification. The language includes three sublanguages: the object definition language (OASIS-ODL), the object query language (OASIS-OQL), and the object manipulation language (OASIS-OML). This identification of sublanguages is based on the ODMG standard [CB97]. The OASIS language has been defined for research purposes; it was not meant to be a full-fledged implementation language. A type-checker has been implemented as part of this research project (see Chapter 6). However, there is (as yet) no compiler that maps to an existing database platform. Throughout this chapter, a single example is used to illustrate the various features of the OASIS language. The example is taken from a real-life case study, called SEPIA ([EFK$^+$96]). The case study concerns a database to structure hypermedia documents, with atomic nodes, composite nodes and links. A complete description of this case-study is found in Appendix A. The following sections present the three sublanguages and give examples based on the SEPIA schema.

3.1 The object definition language: OASIS-ODL

In this section, we introduce the OASIS object definition language, OASIS-ODL (ODL abbreviates object definition language). OASIS-ODL is used for describing the *database schema*. A schema consists of class definitions (including methods and constructors), persistent roots, integrity constraints and atomic transactions. Details of these are given below.

3.1.1 Classes, types and inheritance

Central to OASIS, as in any object-oriented programming language, are the notions of class and object. Similar to Java, OASIS supports both abstract and concrete classes. Here is the class hierarchy of the example database in OASIS, which defines atomic contents, elements, nodes, atomic nodes, composite nodes and links:

```
class AtomicContents {
  attribute string referenceDirectory;
  attribute string showStatement;
  attribute string URL;
  ...
}
abstract class Element {
  attribute string name;
  attribute int position;
  abstract boolean isConnectedTo(Element n);
  ...
}
abstract class Node extends Element {
  attribute set⟨Link⟩ incomingLinks;
  attribute set⟨Link⟩ outgoingLinks;
  boolean isConnectedTo(Element n) { ... };
  ...
}
class ANode extends Node {
  attribute set⟨string⟩ content;
  ANode(string s, int p) { ... } ;
  ...
}
class CNode extends Node {
  attribute int size;
  attribute set⟨Element⟩ elements;
  boolean removeElement(Element n) { ... };
  ...
}
class Link extends Element {
  attribute Node from;
  attribute Node to;
  boolean isConnectedTo(Element n) { ... };
  ...
}

name set⟨CNode⟩cnodes;
name set⟨ANode⟩anodes;
name set⟨Link⟩links;

constraints {
  c1: forall c in cnodes : c!=nil;
  c2: ...
}
```

OASIS is a typed language. The type of an attribute may be an *atomic type* or a *complex type*. Atomic types are the basic types int, bool, char, and string; complex types are constructed using the atomic types, *reference types* (i.e., names of classes defined in the schema), and the type constructors list (for ordered collections), set (for unordered collections), and struct (for labeled records). The use of reference types in attribute declarations gives rise to (mutual) recursive data structures, e.g., Node refers to Link and vice versa.

OASIS supports single inheritance. In the above example, ANode and CNode are *extensions* of class Node, which in turn is an extension of Element. ANode and CNode extend both the *data* and *behavior* of the Node superclass. ANode extends the data by adding the field content. CNode adds two attributes: size and elements. Both the ANode and CNode class also extend behavior by defining specific methods. For example, the CNode class defines a method removeElement. The implementation of this method (not shown yet) using OASIS-OML, is discussed in Section 3.1.2.

Both the Node and Element class are so-called *abstract classes*. A class can extend an abstract or a normal class. The concept of abstract classes is a common feature of object-oriented programming languages (e.g., see [AG96, AC96]). An abstract class is different from a normal class in that it cannot be instantiated. Abstract classes are used for type generalization. For example, a Link can connect two ANode or CNode objects. Such an or-type is conveniently modeled using the abstract class Node. An abstract class cannot extend a normal class. Methods in an abstract class can be declared as *abstract methods* (e.g., the isConnectedTo method in the abstract class Element). Abstract methods are discussed in more detail in Section 3.1.3.

A CNode or an ANode object can be used at any point where a Node or an Element is expected. For example, the from field of a Link can be filled in with a CNode, or an ANode object. This characteristic of objects is known as *subsumption* (or *polymorphism*)—a single object like an instance of CNode can also be used in any place where an instance of one of its supertypes (i.e. Node or Element) is expected.

The extends clause in a class definition is optional. Classes that do not explicitly extend any other class, implicitly extend the most general class Object. In OASIS, the class Object is abstract and thus cannot be instantiated (Object can be used, however, as a reference type like any other name of a user-defined class).

The schema declares three so-called *persistent roots*; i.e., cnodes, anodes, and links. These are the persistent tables of the database. We discuss persistent roots at length in Section 3.1.5. Restrictions on the contents of the persistent roots are specified declaratively in the constraints section (see Section 3.1.6).

3.1.2 Methods

Methods definitions are given using the OASIS object manipulation language (OASIS-OML). OASIS-OML is a simple procedural (i.e., command or instruction) language for the definition of updates on complex objects. The following method is defined in the CNode class, and will be used as a running example in this thesis.

```
boolean CNode::removeElement(Element n) {
    if (n in elements and (forall e in elements: not(e.isConnectedTo(n)))) then {
        elements −= set(n); return true
    } else return false
}
```

This method takes one explicit parameter n of type Element, and returns a boolean value. If a method does not return any value, its 'return type' is void. As a matter of convention, we will often prefix method names with the name of the class in which they are defined. Thus, CNode is the class in which the above definition is given. This avoids repeating the entire class definition in examples.

The method removeElement removes one of the elements (i.e., Node or Link) from the elements of a given CNode object, provided that that element occurs and there are no other connections (within the same CNode object) to that element. The method returns true if the condition is met; otherwise false is returned.

The 'elements'subexpression is actually a shorthand for 'this.elements', where 'this' stands for the receiver object. Conventionally, 'this' should only be added when the attribute name is hidden by a variable (or parameter) declaration with the same name. The condition in the if-part is a complex boolean expression, which involves universal quantification over a set and the application of an abstract method (i.e., isConnected as defined in Section 3.1.3). Such complex conditions are expressed using the object query language OASIS-OQL. Details of this language are discussed in Section 3.2.

3.1.3 Abstract methods and retrieval methods

Methods in an abstract class do not necessarily have an implementation. Each method that is declared but not implemented by the abstract class should be prefixed with the keyword abstract. Further, any concrete class that extends an abstract class should provide an implementation for all abstract methods it inherits from superclasses. Consider, for example the method isConnectedTo, which is used in the definition of the method removeElement mentioned above. The method is declared as an abstract method in the abstract class Element:

```
abstract boolean Element::isConnectedTo(Element n);
```

An implementation, with the same signature (i.e., name and input/output types), should be defined for all concrete classes that inherit from Element, namely for the classes ANode, CNode and Link. We define different implementations for Link and Node (i.e., for both ANode and CNode). Here is the implementation for the abstract class Node:

```
boolean Node::isConnectedTo(Element n) {
   return (n in incomingLinks) or (n in outgoingLinks)
}
```

The implementation for Link is very different, although the signature should not change. Here is the code for Link:

```
boolean Link::isConnectedTo(Element n) {
  return (from == n) or (to == n)
}
```

The isConnectedTo method can be invoked for any object of type Element. In this case, the actual *run-time type* of the receiver object determines which implementation of the method is used. This mechanism is called *late binding*. For example, in the definition of method removeElement, the method isConnectedTo is applied to each object in the collection of elements. Depending on the actual run-time type of the Element at hand (i.e., ANode, CNode, or Link), the right implementation of isConnectedTo is applied.

Methods such as the isConnectedTo method above are called *retrieval methods*. Retrieval methods are methods that do not update any objects. The body of a retrieval methods consists of only a single return statement, where the return value is specified by an OQL expression. These expressions are side-effect free (see Section 3.2). Retrieval methods can be invoked in queries (see Section 3.2), methods that perform updates cannot.

An extended class may add new methods, or it may *override* some of the methods inherited from one of its superclasses. A method defined in a superclass is redefined in a subclass by defining a method with the same name and the same number of parameters and parameter/return types as specified in the superclass. Again, late binding determines which implementation should actually be used.

In Java it is also possible to *hide* some of the attributes from a superclass. Hiding an attribute means that if we declare an attribute in a class and that attribute has the same name as another attribute in one of the inherited superclasses, then the inherited attribute is hidden and can only be accessed using the keyword super, instead of this. In OASIS, this feature has not been taken into account.

3.1.4 Constructors

In OASIS-OML, new objects are created using the command new followed by the class name, e.g. the command

```
new ANode
```

creates a new ANode object and assigns default values to its attributes (see Section 3.3.1). Sometimes this is sufficient to ensure a correct initial state of the object. Often, however, the piece of code that creates the object needs to supply initial data to one or more of the attributes of the object. For these purposes, classes can define *constructors*.

Constructors are like methods. They take zero or more parameters, but their name is always identical to the name of the class in which they are defined and (unlike methods) there is no return type. For example, our schema defines the following constructor for the class ANode:

```
ANode(string s, int p) {
  name = s;
  position = p
}
```

The body of the constructor is applied to the new object after the default values have been assigned. The above constructor takes two arguments, which are used to initialize the name and position field. The other set-valued attributes (i.e., incomingLinks, outgoingLinks, and content) maintain their default value (i.e. the empty set value set()). At present, there is no overloading of constructors (like in Java). Hence, classes can have at most one constructor.

3.1.5 Persistent roots

Objects and values do not automatically remain in the database after the application that created them has terminated. OASIS adopts the persistence approach of the O2 system, which is often called *persistence by reachability*, or *transitive persistence* ([BDK92]). In this approach, the schema defines a number of named entry points to the database (similar to tables in the relational data model). These entry points act as global variables of the schema (i.e., they can be used in the definition of methods, transactions and constraints); all objects and values directly or indirectly reachable from these names are persistent. No other objects or values are persistent.

For the example schema, a persistent root cnodes is defined, which is a set used to store instances of the CNode class:

```
name set⟨CNode⟩cnodes
```

A CNode object is made persistent by simply inserting the object into this set, e.g. the OASIS-OML command

```
cnodes += set(c)
```

provides persistence to a CNode object c *as well as all objects it references*. Observe that the cnodes name behaves as a normal (global) variable, the only difference being that its contents are persistent.

The relational data model only provides persistence to relations. In OASIS one can also define persistent constants, or provide persistence to a single object, e.g.

```
name int nrOfnodes
name Node top
```

The name top can, for instance, be initialized with an instance of CNode or ANode, or with the nil literal (remember that class Node itself cannot be instantiated because it is abstract). The name gives persistence to the object, and it can be used as a global variable in method definitions, constraints, and transaction programs.

3.1.6 Integrity constraints

OASIS is a typed language; all objects and values have a type. Type information in the schema enforces relatively simple constraints on the data. Additional, more complex constraints can be expressed in the constraints section of the schema. A constraint is an arbitrary boolean-valued OASIS-OQL expressions over the *database state* (i.e., persistent roots), which includes universal and existential quantifications, and nil comparisons.

For the example schema, a constraint c1 is defined

```
c1 : forall c in cnodes: c!=nil
```

which asserts that nil-references cannot be inserted in the persistent root cnodes. Since the constraint can be an arbitrary OASIS-OQL expression, more complex constraints can be expressed as well. The following constraint asserts that any Link inside a CNode only connects elements of that same CNode:

```
c1 : forall n in cnodes :
       forall e in links: (e in n.elements) implies
         ((e.from!=nil implies e.from in n.elements) and
          (e.to!=nil implies e.to in n.elements))
```

Constraint definition in OASIS has its limitations. In particular, aggregate constraints involving collection operations such as count and sum are not supported yet, see Section 3.4. Integrity constraints should hold for all objects in the database, between transactions. Constraints may be temporarily violated during transaction execution as long as they hold at transaction commit. Programmers should normally include tests in method and transaction code that prevent integrity violations to occur. This technique is often called *defensive programming*. A major topic of this thesis is the development of a verification tool that assists database designers to verify—at compile-time—the correctness of a transaction/method with respect to the constraints specified in the schema.

3.1.7 Transactions

Transactions are similar to methods. They are used to update and/or query the database contents. Definitions of transactions are given in the same way as methods, using OASIS-OML. A transaction may invoke methods, but transactions may not be invoked from other transactions or methods. Another important difference between a method and a transaction is that the latter does not use the this parameter; transactions apply to an entire database state rather than a single object. The following example transaction clearly illustrates this difference

```
void removeAtomicContent(string ac) {
  foreach a in anodes where (ac in a.content) do
    a.content -= set(ac)
}
```

The above transaction removes an atomic content, from those elements of the persistent table anodes that satisfy the condition specified in the where-clause.

Notice the use of the foreach-operator, which is different than forall; i.e., foreach is an collection iterator used to program updates, unlike forall which is a logical quantifier that returns a boolean value. The foreach loop allows for the definition of set-oriented updates. As pointed out in [LS93], it is well-known that these are problematical in the context of object-sharing. Section 3.3.8 details on these problems and how they are dealt with in OASIS.

3.2 The object query language: OASIS-OQL

OASIS-OQL is a subset of OQL, the object query language of the ODMG standard [CB97], and the O2 database system [BDK92]. OASIS-OQL is a read-only query language without side-effects. It is a typed expression language that deals with complex values and objects. In contrast to SQL2 [SQL92], a query may not only return a flat relational table, but it can also return arbitrarily nested data structures with object references. In this section, we informally discuss the various language constructs available in OASIS-OQL. A complete formal definition of the language, including abstract syntax and typing rules is given in the appendix.

3.2.1 Variables

Variables can be used in query expressions. Global variables are the ones declared as persistent roots in the schema. For example, anodes is a valid expression of type set$\langle$ANode$\rangle$ in the context of the example schema. Additional variables are typically introduced by collection iterators, such as select-from-where and the quantifiers exists and forall.

3.2.2 Atomic type literals

Atomic values have the usual syntax:

- Integers: e.g., -24 or 200,
- Boolean values: true, false,
- Character values: Character values appear between quotes (e.g., ‘z‘) ,
- Strings: A string literal appears between double quotes (e.g., “John”),
- Objects: The only literal object reference is nil. It can be used anywhere where a reference is expected. The nil literal is used for an invalid or uncreated object. Its type is a wild-card that matches all class names (see Section 3.2.5).

3.2.3 Constructing complex values

Complex values are constructed using the constructors list, set, and struct:

- Record: The expression struct$(a_1 : e_1, \cdots, a_n : e_n)$ represents a record structure with n fields. For example, the expression struct(name:"John",age:25) is a record with two fields name and age with values "John" and 25, respectively. The type of this expression is written as struct(name:string,age:int).

- List: The expression list$(e_1, \cdots, e_n)$ represents a list expression with n elements. For example, the expression list(10,20,10) is a list of three integer values. The type of this expression is written as list$\langle$int$\rangle$.

- Set: The expression set$(e_1, \cdots, e_n)$ represents a set expression with n elements. For example, the expression set(10,20) is a set of two integer values. The type of this expression is written as set$\langle$int$\rangle$. Duplicates are automatically removed from the set.

New objects cannot be constructed in the OASIS-OQL language; object creation is dealt with using the language of commands, OASIS-OML (see Section 3.3).

3.2.4 Heterogeneous collections and type compatibility

The elements of a list or set need not be of exactly one and the same type. For example, if n1 and n2 are expressions of type CNode and ANode respectively, then the expression set(n1,n2) is correctly typed and of type set$\langle$Node$\rangle$. Such a set or list with slightly differently typed components is often called a *heterogeneous collection* (e.g., see [vK97]) or a *polymorphic collection* ([CB97]). As pointed out in [CB97], a major contribution of object-oriented programming is the possibility to manipulate heterogeneous collections of objects and, thanks to the *late binding* mechanism, to carry out generic actions on the elements of these collections.

The OASIS language does not allow values of arbitrary types to be grouped in one and the same collection. Values can only be put in the same collection as long as they have *compatible types* ([CB97]). For example, the types CNode and ANode in the above example are compatible because of subsumption: any instance of class CNode or ANode is also of type Node. Node is called the *least-upper bound* (LUB) of the types CNode and ANode; it provides the most detailed information about the properties of state and behavior shared by both ANode and CNode. The (existence of a) LUB is a requirement for the formation of many OQL expressions (see Appendix C). The LUB of two reference types always exists, because they are all subtypes of the most general class Object. Hence, objects can always be put in one and the same set.

Type compatibility is not merely defined for reference types. One can also construct heterogeneous collections of complex values. In this case, compatibility boils down to the compatibility of the atomic components. For example, set$\langle$CNode$\rangle$ and set$\langle$ANode$\rangle$ are compatible, and so are struct(n:CNode) and struct(n:ANode) but *not* struct(n:CNode,l:Link) and struct(n:ANode). Hence, records with a different number of components are not compatible. Note that this is in contrast to the various forms of record polymorphism, discussed in [CW85, Car88]. For more details, the reader is refered to Section 2.1.

3.2.5 Constants without a type

OASIS-OQL is a strongly typed language, in which each (correctly typed) expression is assigned a unique minimal type. Some expressions are assigned a wild-card type (written $*$). These are the expression set(), which is the empty set of type set$\langle * \rangle$, and the expression list(), which is the empty list of type list$\langle * \rangle$. The type $*$ is compatible with any other type, including reference types. Similar to this is the type Nil, which acts as a wild-card for reference types; this type is used to type the nil literal. The type Nil is compatible with any other reference type, e.g. Nil is compatible with Link.

The wild-card and Nil type cannot be used by schema designers in the signature of methods, attribute and variable declarations. These types are only used to get around the typing of the above mentioned literals.

3.2.6 Arithmetic expressions

Arithmetic expressions are the usual ones for addition ($+$), subtraction ($-$), multiplication ($*$) and division ($/$). The types of the operands of these operations should be integer values. The operation $+$ is overloaded and can also be used for list and string concatenation (as discussed in Section 3.2.10).

3.2.7 Boolean expressions

OASIS-OQL defines the usual infix overloaded operations for equality (==), and its negation (!=). Typing requirements enforce the existence of a least-upper bound (LUB) for the type of the operands. For example, if cnode is an expression of type CNode, and node an expression of type Node, then the expression

```
anode == node
```

is correctly typed and of type bool, because reference types are always compatible (see Section 3.2.4). Hence, we could even compare an expression of type Person with one of type CNode. This may seem to be awkward: how could they possibly denote the same object? The answer is simple: such a comparison makes sense—and evaluates to true—if both expressions denote the nil literal. The OASIS mini-type-checker that has been implemented generates a warning message if two references are compared under the most general type Object; this case often indicates a typographic error in the schema.

Set membership (in) is an overloaded construct that works for lists and sets. In this case, a LUB should exist for the type of the operand on the left-hand side and the type of the elements of the collection on the right-hand side. For example, if anode is a variable of type ANode and nodes a named root of type set$\langle$Node$\rangle$, then the following expression is correctly typed and of type bool:

```
anode in nodes
```

The operations '$<, >$', '$<=$' and '$>=$' are defined only if both arguments are numbers (int) or sets. For sets, the same typing restrictions apply as for the equality operator: a LUB should exist. For example, if cnodes is a collection of type set⟨CNode⟩, and nodes a collection of type set⟨Node⟩, then the expression

```
cnodes <= nodes
```

is correctly typed and of type bool. Between sets, the operation $<=$ is interpreted as $\subseteq$. Again, we could write 'weird' code, e.g. cnodes $<=$ People.

The expression not(e) is used for the negation of e. Boolean connectives are the infix operations and, or, and implies. Existential and universal quantifiers can be used. They are always bound by a collection (i.e., list or set); unbounded quantifiers cannot be used. Bounded quantifiers are particularly useful for defining constraints on the database (for examples, see section 3.1.6).

3.2.8 Path expressions

The '.' operator is used to select an attribute of an object or a record structure. Thus so-called path-expressions (i.e., expressions which involve nested application of the '.' operator) can be formed to navigate through a complex object or record structure. The following example expression selects the position of the Node that is attached to the from field of a Link object 'l'.

```
l.from.position
```

If the Link object turns out to be the nil literal, then the result of the attribute selection is *undefined* as described in Section 3.2.14.

3.2.9 Collection expressions

Various collection operations are defined. These are operations that work for arbitrary collection types (in our case lists and sets).

OASIS-OQL uses the familiar select-from-where statement for collection iteration. The following expression builds a record structure for all CNode objects from the named root cnodes, which are positioned in the interval $[175, 1500]$; the structure that is built contains the name of the CNode, paired with the name of its elements

```
select struct(name: c.name,
              elms: (select n.name
                     from n in c.elements))
from c in cnodes
where (c.position>175) and (c.position< 1500)
```

The result of this query is of type set⟨struct(name:string,elms:set⟨string⟩)⟩. In ODMG-OQL, the result of the above expression is a bag (with the same type of elements). To return a set, an extra distinct clause needs to be inserted right after select. In this thesis, bags are not discussed and the distinct clause can be omitted.

Observe that nesting may occur at arbitrary places: unlike in SQL, operations such as select-from-where can be freely composed as long as simple typing constraints are preserved.

Other generic operations on collections are element, and flatten. The element operator extracts the unique element of a singleton collection. The result is undefined if no single element exists, e.g.

```
element(set(10))
```

is of type int and evaluates to 10.

The flatten operator works on a collection of collections (i.e., on an expression e of type col⟨col⟨t⟩⟩), where col abbreviates list or set; unnesting is applied by unioning the elements of e. The result type depends on the kind of collections that are used: if one of the collection types is a set, then the result is of type set⟨t⟩; otherwise it is of type list⟨t⟩. The following example is taken from [CB97] and returns the set containing the elements $1, 2, 3, 4, 5, 6,$ and 7 (duplicates are removed):

```
flatten(list(set(1,2,3),set(3,4,5,6),set(7)))
```

3.2.10 List & string expressions

In addition to the operations that work for arbitrary collection types, there are also a few specific operations for lists. These operations are overloaded for strings (a string is treated as a list of characters).

The operators head and tail are used to extract the head and tail of a list or string. Both operators have an *undefined* case; i.e., the result of head(list()) and tail(list()) is *undefined*.

A list is converted to a set using the 'listtoset' operator. Such a conversion looses the ordering of the list, and removes any duplicates. For example,

```
listtoset(list(2,7,8,7))
```

returns the set containing the elements 2, 7, and 8.

List (or string) concatenation is performed using the infix operation '+'. e.g. the following expression results in the list of four elements list(1,2,2,3).

```
list(1,2) + list(2,3)
```

3.2.11 Set expressions

Specific operators on sets are the usual infix operations for intersection (intersect), set-union (union), and set-difference (except). For example, if nodes is a collection of nodes and cnodes a collection of composite nodes, then the expression

```
nodes except cnodes
```

evaluates to the set of nodes which are not in cnodes. The type of this expression is the type of the set on the left-hand side, i.e., set⟨Node⟩. Set-difference is well defined if a least-upper bound exists of the type of the operands.

The result of the union or intersection of two sets is the least-upper bound of the type of the operands. For example, the expressions

```
cnodes union anodes
cnodes intersect anodes
```

are both of type set⟨Node⟩. One could argue that a greatest-lower bound should be used for typing the set intersection operation (see e.g. [vK97]). We have decided, however, to follow ODMG-OQL typing rules, which uses a LUB for all binary operations on sets.

3.2.12 Method application in queries

Inside a query one can invoke methods. Methods are applied to a receiver object in the same way as we select an attribute of an object, using the '.' notation. The type of the value returned by the invocation of a method is the return type specified in the method signature. For example, given an Element object 'e' and Node object 'n', the following expression is correctly typed and of type bool

```
e.isConnectedTo(n)
```

where the method is defined as in Section 3.1.3. The receiver object and the actual parameter values can be subtypes of the ones specified in the signature of the method. For instance, we could also apply the above method to a receiver of type ANode and a parameter of type CNode.

OASIS-OQL is a query language without side-effects. All updates to the database are expressed using a separate update language, OASIS-OML. Methods, however, may update the database state. To prevent such *hidden updates* to occur inside a query expression, the syntax is restricted such that only methods can be applied which have a body that consists of a single return statement (see Section 3.1.3 for details).

Similar to attribute selection, if the receiver object turns out to be the nil literal, then the result of the attribute selection is *undefined* as described in Section 3.2.14.

3.2.13 Late binding

The OASIS data model supports subsumption. Wherever an expression of a certain type is expected one can always use an expression of a subtype. For example, if we have a collection of static type set⟨Element⟩, then the elements of this collection can be instances of ANode, CNode or Link. OASIS-OQL allows us to invoke abstract or overridden retrieval methods to the elements of such heterogeneous collections of objects, using *late binding*. For example, in Section 3.1.2, we introduced the abstract method isConnectedTo for class Element. If we apply this method to an object of type Element, the actual run-time type of the object determines which implementation of the method is used. For example,

```
select e
from n in cnodes, e in n.elements
where e.isConnectedTo(node)
```

the above query selects from all cnodes those elements that are connected to a named Node object node. The result of this query is of type set⟨Element⟩ (because Element is the type of e).

In OASIS, a single-dispatch model is used, in which only the type of the receiver object determines the actual binding. This is conform the ODMG standard and many object-oriented programming languages (e.g., Java).

Notice that the expressions in the from clause do not need to be 'independent' collections. As is seen in the example above, a collection in the from part can be derived from a previous one by following a path that starts from it.

3.2.14 Undefined values

OASIS supports nil values. The result of attempting to access the attribute of, or application of a method to a nil reference is *undefined*, which in OASIS means that the semantics does not prescribe the outcome (in fact, an arbitrary value is chosen). Ideally, an exception should be raised to capture such cases, but exceptions are not delat with in this thesis.

3.2.15 Type casting

Sometimes, static type checking fails to infer that an object is actually of a more specific type. In those cases, one can use an explicit down-cast of the object to its right type, e.g.

```
select (Link)e.from
from n in cnodes, e in n.elements
where (e in links)
```

The above query selects the elements of all CNode objects. The selection is restricted to Link objects only (by the condition specified in the where-clause). By down-casting the elements to Link, we obtain their from field. If the down-cast fails (i.e., the selected element turns out to be not a Link), the result is *undefined*.

3.2.16 Instanceof

Related to type casting is the instanceof operator, which is used to determine if an expression if of a given type. The meaning and syntax of this operator is the same as in Java. For example, consider an object e of static type Element. To check whether e is actually of type Node (thus not of type Link), we use the following expression:

```
e instanceof Node
```

The above expression returns true if the down-cast of e to Node would be valid, otherwise false is returned (this is also the case if e turns out to be the nil literal).

3.3 The object manipulation language: OASIS-OML

OASIS defines a separate update language OASIS-OML for object manipulation. OASIS-OML is a superset of OASIS-OQL: any OASIS-OQL query expression is also a valid command (in this case, no side-effects occur). A small set of primitive commands is defined (e.g., for object creation, attribute update and variable update). There is no object deletion because of the persistence by reachability that is used. Compound updates are formed using sequential composition (;), conditional branch (if-then-else), (non-recursive) update method call, and collection iteration (foreach). A command may also return a value (i.e. a query result). In case no value is returned, the command is of type void.

3.3.1 Object creation & default values

New objects are created using the new command. For example, the following command creates a new instance of class ANode:

```
new ANode
```

The type of this command is ANode: 'new' returns a reference to the new object. The attributes of the object are automatically initialized. This is done by assigning a *default value* to each of the object's attributes. The rules for assigning default values are as follows:

- Integers are assigned the value 0
- Characters are assigned the null character \0000
- Strings are assigned the empty string ""
- Booleans are assigned the value false
- Sets are assigned the empty set value set()
- Lists are assigned the empty list value list()
- Structs are (recursively) assigned default values for each of the attributes
- Object references are assigned the nil literal

Sometimes, assigning default values is sufficient to ensure a correct initial state of the object. But often more sophisticated data initialization should be performed. For these purposes, OASIS supports constructors, as discussed in Section 3.1.4.

In Section 3.1.4, we defined a constructor for ANode with two parameters. The new command supplies the actual values for these parameters.

```
new ANode(“X105”,575)
```

The above command creates a new instance of ANode; after its attributes have been assigned their default values, the constructor is applied (the constructor initializes the name and position field).

Static type checking validates whether the number and types of the parameters supplied to new correspond to the ones declared in the signature of the constructor. If there is a constructor but the number and/or types of parameters do not match, a type checking error is generated. There is only one exception to this rule: if a constructor with parameters has been defined, but no arguments are supplied to new, then only default initialization is performed.

3.3.2 Assignment

Attribute values or variables are assigned a new value in the usual way, using the assignment operation '='. The type of an assignment is always void, unlike in Java. Because of polymorphism, the new value may actually be a subtype of the (declared) type of the attribute or variable that gets the new value. For example, let l be a variable of type Link, and cnode be a variable of type CNode, then the following assignment is correctly typed and of type void:

```
l.from = cnode
```

In the above example, the new value is determined by the query expression cnode. But the new value can also be determined by a return value of a command. A typical example is the binding of a reference returned by new to a variable, e.g.

```
n = new ANode("X105A",575)
```

Related to the assignment operator, are the operators '+=' and '-=', for *incremental and decremental assignment*, respectively. These operators can be regarded as convenient shorthands for a normal assignment. *Incremental assignment* 'x+=c' executes the command 'c', and subsequently 'adds' its result to the current value of 'x'. For adding the result, the operator '+' is used in case 'x' is of type int or list; in case 'x' of type set, the set-union operation ('union') is used. Typing restrictions enforce c to be a subtype of the type of x. *Decremental assignment* works in a similar way; the subtraction operation is '-' in case x of type int or list, and 'except' in case of sets. In this case, typing restrictions allow c to be of any type that is compatible with the type of x.

3.3.3 Deleting an object

OASIS uses persistence by reachability. An object remains in the database as long as the object is (directly or indirectly) reachable from at least one of the named roots. Consequently, to "remove" an object from the database, all references to the object—reachable from the roots—have to be "removed". This is done using standard OASIS-OML statements. For example, to remove an ANode n from the root cnodes, one uses:

```
cnodes -= set(n)
```

An individual reference can be broken by assigning the nil literal to it, or by replacing the reference by a reference to another object, e.g.

```
top = nil
```

3.3.4 Return

Object creation is a typical example of a command that combines operational and functional behavior. Many other commands do not return a value (i.e. they are of type void). In this case, an explicit return value is specified using the return e statement, which returns the value denoted by the OQL expression e. It is important to know that return is not a control flow statement, like for instance the return statement in Java. Hence, it does not terminate execution of a method.

3.3.5 Sequential composition

Atomic updates can be sequentially composed to construct non-trivial updates, e.g.

```
n = new ANode(s, p);
elements += set(n)
```

It should be observed that commands cannot be mixed with query expressions. In OASIS, one *cannot* use the following piece of code instead of the above explicit sequencing of commands:

```
elements += set(new ANode(s, p))
```

The reason why the above expression is not allowed is that set is a query language construct (see Section 3.2.3), which expects a query (not a command) sub-expression.

The type of a command sequence $c_1; c_2$ is the type of c_2. For example, the above command sequence is of type void because the assignment '+=' is of type void. The following command sequence is of type Link (provided that n is of type Link):

```
elements += set(n); return n
```

3.3.6 Local variables

Local variables are declared in a straightforward manner, e.g.

```
Node n = new ANode(s, p);
elements += set(n)
```

The above command declares a new variable n of type Node. Local variables are *not* assigned default values. Instead, each variable has to be initialized explicitly (as in Java). In the above example, the initial value of n is the reference returned by the new statement.

3.3.7 Conditionals & skip

Conditional updates are performed using if (b) then c_1 else c_2. The expression b is an arbitrary boolean-valued OASIS-OQL expression (which may include quantifiers), which is evaluated first. If its value is true, then c_1 is executed, else c_2. Both b, c_1 and c_2 observe the same incoming database state. The type of the value that is returned is the least-upper bound of the types of the then and else branch; a type checking error occurs if no such bound exist, e.g.

```
if (n in elements) then {
   elements -= set(n); return n
} else return nil
```

The above statement removes an Element n from the set of elements and returns n, if the former is a member of the latter; otherwise, no operation is performed and nil is returned. The type of the "then" branch (namely and Element) is compatible with the type of the "else" branch (namely Nil). The result type is the least-upper bound of Element and Nil, which is Element.

The else clause is *not* optional. A command called skip should typically be used if no operation is to be performed, e.g.

```
if (n in elements) then {
   elements -= set(n)
} else skip
```

The above statement removes an element n from the sets of elements, if the former is a member of the latter; otherwise, no operation is performed. Both branches are of type void.

3.3.8 Bounded iteration

A simple procedural language with atomic updates such as assignment and object creation, combined with sequential composition and conditional updates is not sufficient. Many database programming languages proposed in the literature use some form of bounded-iteration construct or recursion, to gain 'realistic' expressiveness. OASIS uses bounded iteration, recursion is not supported. Here is a simple example, that shows how collection-oriented updates are expressed in OASIS-OML:

```
foreach k in n.incomingLinks where (k.name = "X124") do {
    k.disconnectLinkFromNode(n)
}
```

The above command iterates over the set of incomingLinks, and applies the disconnectLinkFromNode method to each element in the set. The iteration can be bound by a list or a set. In the case of a list, the collection is first converted to a set before the actual iteration starts. The command body of the iteration can be an arbitrary command (including sequential composition, and other iterations). The where clause is optional.

It is well known (e.g. [LS93], and [vK97]), that conflicts may arise with set-oriented updates due to *object sharing*: multiple incompatible updates can be applied to one and the same object in different steps of the iteration. The following example illustrates the problem in OASIS-OML.

```
Node n1 = new ANode(“X125”,275);
Node n2 = new ANode(“X126”,150);
lnk1 = new Link(“Y875”);
lnk2 = new Link(“Y876”);
lnk1.from = n1;
lnk2.from = n1;
lnk1.to = n1;
lnk2.to = n2;
foreach k in set(lnk1,lnk2) do {
   k.from.position = k.to.position∗2
}
```

What happens if the above update is applied? If a sequential processing of the elements of the set is used, the effect of the update is *non-deterministic*: depending on the order in which we take the elements from the set, the shared Node in the from field gets the new value 300 or 600 for its position field. In OASIS, we have decided to use *parallel* execution semantics for the bounded-iteration construct; i.e., each step in the iteration is executed in the same incoming database state, and no intermediate states are observed. This provides downwards compatibility with SQL2's update-from-where statement (see [SQL92]). Compared to a sequential processing of the loop, parallel execution semantics “reduces the complexity of computation and reasoning about the effects of program executions on the database contents” (see [Qia93]). Intuitively, this can be understood by the fact that parallel execution semantics works collection-at-a-time, as opposed to the tuple-at-a-time processing implied by a sequential processing of the loop.

Parallel execution semantics, however, does not solve the problem of conflicts in the definition of set-oriented updates; e.g., using parallel execution semantics, the Node object n1 in the above example is assigned a new position of 550 and 300 at the same time. OASIS does not support an exception mechanism to deal with such cases. Conflicts of multiple updates to the same object are “solved” by treating the conflicting cases as no-ops (i.e. in case of a conflict, no update is performed). Hence, in the above example, the object n1 is not updated at all. Ideally, an exception should be raised.

Variables declared outside the scope of the iteration cannot be updated inside the loop. This would likely raise conflicts, similar to the ones mentioned above: a variable cannot be assigned different values in parallel. Parallel execution semantics leads to a slightly different (i.e. more declarative) style of programming loops. Here is an example of programming a loop construct in O2C [BDK92] (which uses sequential processing of the elements of the collection):

```
o2 set(Link) s;
```

```
s=set();
for (x in set("Y875","Y876","Y877") {
     s+=set(new Link(x))
}
```

The variable s declared outside the loop is used to accumulate the newly created Link objects in a set. In OASIS, one can program such a loop in a different way. Here is the OASIS code for the same example:

```
set⟨Link⟩ s = foreach x in set("Y875","Y876","Y877") do { new Link(x) }
```

In fact, the foreach-loop combines operational and functional behavior. The values returned by the different steps of the iteration are automatically accumulated in a set; if no values are returned (i.e. the command body is of type void), the result of the evaluation of foreach is also of type void.

3.3.9 Method call

Methods are invoked in a similar way as we access an attribute, using dot (.) notation. Actual parameter values are passed by using a comma-separated list enclosed by parentheses. OASIS uses *call-by-value* for parameter passing, similar to Java. That is, all values passed for parameter variables are treated as copies. Thus, changes applied to parameter variables do not affect the values in the environment of the caller. When the parameter is an object, however, only the reference is passed. Changes applied to *attributes* of such an object will be visible in the environment where the method was invoked. Thus side-effects can occur. For example, consider the following definition of a method in class Link.

```
void CalByRef(Link p) {
  p.name = "X100";
  p = nil
};
```

The update of the name field applied to the parameter ‘p’ is visible in the environment that invokes the method. The update of ‘p’ to the nil reference, however, is not observed in the environment of the caller. For example, consider the following command sequence:

```
Link x = new Link;
x.name = "X125";
x.changeName(x);
return x.name
```

The result of executing the above command sequence is the string value “X100”. The value of ‘x’ is not affected by the invocation of the method (for otherwise we could not select its name field in the return statement).

Late binding is used if an abstract or an overridden method is invoked (see Section 3.2.13).

3.4 Summary

An object-oriented database programming language, called OASIS, was discussed. OASIS supports many of the common features of current object-oriented database programming, such as heterogeneous sets, nil values, and late binding of method calls. In addition to these, there are a number of other relevant features have not been included. These are (just to mention a few): *bags* and *arrays*—as yet only one ordered collection (i.e., list) and one unordered collection (i.e., set) are supported, *aggregates* (e.g., operations such as count and sum), specialized mathematical functions (e.g., min,max and sqrt), and name overloading where several methods with the same name—but a different number of parameters—can be defined in the same class. The reason why these features have been omitted in OASIS is twofold.

First of all, a major goal of this dissertation is to build a research prototype to demonstrate that verification of object-oriented database specifications using a modern higher-order logic theorem prover is feasible. Part of such a research prototype is the implementation of a compiler that translates an OASIS schema to higher-order logic Isabelle code. Implementation of some of the above mentioned topics (i.e. attribute hiding and name overloading within the same class) address standard compiler issues, but will not introduce any real new issues during proofs.

Secondly, it has *not* been a goal of this dissertation to do research in mathematics. The standard distribution of Isabelle only comes with a theory of sets and lists; other collection types often found in object-oriented databases[1] are not fully supported yet. Also, the Isabelle system does not support aggregations over sets. Although we realize their importance for databases (in particular for constraint specification), we feel that it would be distracting to try to provide too many extensions to the theorem prover. We have decided to leave these issues for future work.

[1]The ODMG standard supports arrays and bags in addition to lists and sets

Chapter 4

Higher-order logic theorem proving using Isabelle

This chapter introduces the higher-order logic Isabelle theorem prover. There is an extensive amount of literature on both the formal and practical aspects of this system (e.g., see [Pau89, Pau90, Pau93b, Pau93a, Pau94, Nip98, Isa]). In this chapter, details are given to the extent that is needed for reading the later chapters of this thesis.

The chapter is organized as follows. In Section 4.1, we explain the main differences between first-order logic and higher-order logic. Section 4.2 then introduces the Isabelle system. It discusses the higher-order logic notation used in the later chapters of this thesis, and the construction of proofs. Section 4.3 discusses an example proof in Isabelle. The example considers a simple proof of an invariant for a transaction on a relational database, using the standard facilities of the theorem prover. The reader is encouraged to read this example: it not only introduces basic theorem proving in higher-order logic, it also motivates the more intricate cases to come for an object-oriented database (as discussed in Chapter 7).

4.1 What is higher-order logic?

Higher-order logic is a generalization of first-order logic, where variables are allowed to range over functions and predicates. There are various kinds of higher-order logic, which may differ in the type system that is used, but the possibility to quantify over functions and predicates is distinctive. Isabelle's higher-order logic theory is inspired by the functional programming language ML, which in turn is based on the simply typed λ-calculus with polymorphic types [Bar84]. For a good introduction of ML and a short introduction of the λ-calculus, the reader is referred to [Pau96].

4.2 Higher-order logic Isabelle theorem prover

Isabelle is a generic theorem prover [Pau94, Nip98, Isa], which supports reasoning in a variety of logics (the so-called *object logics*). The core of the system is a minimal fragment of higher-order logic (the so-called *meta logic*), which is used to represent the syntax and inference rules of the various object logics. Examples of predefined object logics are first-order logic, Zermelo-Fraenkel set theory and (typed) higher-order logic. In this thesis, we will focus on typed higher-order logic. From now on, Isabelle/HOL refers to this particular instantiation of the theorem prover, but often we will simply write HOL or Isabelle.

Isabelle's higher-order logic theory is inspired by the functional programming language ML[Pau96], which may sometimes be confusing. HOL is not a programming language; it merely provides a way to reason about logical theories written in an ML-like style. There is no execution model for HOL. HOL is a specification language, which permits the definition of declarative constructs (such as predicative sets) for which there may not even exist an operational semantics.

The standard HOL distribution of the theorem prover predefines many of the commonly used data types in programming languages. Below, we summarize the syntax of predefined types and terms that will be used later on in this thesis.

4.2.1 Types

The type system of HOL is very similar to that of ML. Basic types include `bool` for boolean values, `nat` for numbers, and `int` for integers. Several type constructors are predefined. Function types are written as $\sigma \Rightarrow \tau$, where "curried" types $\sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \tau$ are often abbreviated as $[\sigma_1, \cdots, \sigma_n] \Rightarrow \tau$. All functions in HOL are total, but partiality can be modeled using optional values (see Section 4.2.2). Other predefined type constructors include product types written as $\sigma \times \tau$, and the collection types `set` and `list`; e.g. $(\texttt{nat})\ \texttt{list}$ represents a list of natural numbers and $(\texttt{nat} \times \texttt{bool})\ \texttt{set}$ a set of tuples where the first field is a number and the second field a boolean. Thus, complex (nested) data structures are supported. This is important for object-oriented databases where similar data structures are found (in contrast to relational databases, where only "flat" table structures are available).

Type schemes provide ML-like polymorphism. For example, we can use types such as the type $[\alpha, \alpha\ \texttt{list}] \Rightarrow \alpha\ \texttt{list}$ for the polymorphic function to add a value of type α to an existing list, where the elements are of type α and α can be instantiated with any other type (including a function type).

4.2.2 Terms

Terms are made up from variables, constants, functions, and function application. Function applications are written in a curried style (e.g., $f\ a\ b$ for the application of the function f to the arguments a and b). Functions are defined using λ notation (e.g., $\lambda x \cdot x + 1$ for the successor function, which is of type $\texttt{nat} \Rightarrow \texttt{nat}$). The composition of two functions f and g is written as $f \circ g$.

Constants are defined for the symbols in the logic (e.g., $+$, $\wedge$, $\forall$). Below, we give an overview of the constants that are predefined by the standard HOL distribution of the theorem prover.

- **Formulae** are terms of type `bool`. They include the basic constants `True` and `False`, and the usual logical connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$, as well as the quantifiers $\forall$, $\exists$, and $\exists!$. Isabelle provides different concrete syntax for these symbols but we will use the more common mathematical notation throughout. For example, we will write $P \wedge Q$ for logical conjunction, rather than $P \& Q$ which is the corresponding concrete Isabelle/HOL syntax. All logical symbols are defined as (function) constants in the logic. For example, `True` is a basic constant of type `bool`, $\wedge$ a function constant of type $[\texttt{bool}, \texttt{bool}] \Rightarrow \texttt{bool}$ and $\forall$ a function constant of type $[\alpha \Rightarrow \texttt{bool}] \Rightarrow \texttt{bool}$. Equality is defined for all terms: $e_1 = e_2$ is of type `bool` provided that e_1 and e_2 have the same type.

- **Conditionals** are written in the usual way, using mixfix notation: $\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ is of type σ provided that e_1 and e_2 are both of type σ and b is of type `bool`.

- **Integer** values include basic constants such as 2, -10. Isabelle predefines the operations $+$, $-$, $\times$, $<$, and $<=$. Similar notation is used for natural numbers, but our focus is on integers.

- **Let** constructs are convenient for abbreviating subterms: $\texttt{let}\ x = e\ \texttt{in}\ e'$ is equivalent to substituting e for all free occurrences of x in e'. Multiple bindings are separated using semi-colons: $\texttt{let}\ x_1 = e_1; \cdots; x_n = e_n\ \texttt{in}\ e'$ as in ML.

- **Set** operations are predefined. Enumerated sets are written as $\{e_1, \cdots, e_n\}$. The usual connectives for sets ($\cup$, $\cap$, and $-$) are predefined. Set-membership is written as $a \in A$, which is of type `bool`. Set-comprehensions are written as $\{x \mid P\ x\}$, which represents the set of all elements satisfying the formula $P\ x$. Functional replacement is written as $\{e\ x \mid x \cdot P\ x\}$, which represents the set of those values $e\ x$, such that $P\ x$. Multiple iterations are written as $\{e\ x_1 \cdots x_n \mid x_1 \cdots x_n \cdot P\ x_1 \cdots x_n\}$. Set-bounded universal and existential quantifications are written as $\forall x \in A \mid P\ x$ and $\exists x \in A \mid P\ x$ respectively.

- **List** is a data type with many predefined operations. For example, $\texttt{hd}\ l$ is used to extract the head of the list l, while $\texttt{tl}\ l$ results in the tail of l. List concatenation is written as $l_1 @ l_2$. In this thesis, our focus is on operations on sets rather than lists (although list operations are equally well supported by Isabelle).

- **Hilbert's ϵ operator** is a declarative operator: $\epsilon z \cdot P\ z$ denotes the unique value satisfying the formula $P\ z$. If no such unique value exists, an arbitrary value is chosen. The special case $\epsilon z \cdot \texttt{False}$ is written as `arbitrary`. The `arbitrary` constant provides a common way of dealing with undefined function results in HOL, which inhabits any types. It is, for instance, used to define the head of the empty list in the standard Isabelle/HOL library. We will use `arbitrary` values to deal with wrongly typed attribute selections (e.g., due to nil-values) and down-casting (see Section 5.7).

- **Option** is a data type that provides an alternative to model partiality in HOL. The option data type is defined as follows:

$$\texttt{datatype}\ \alpha\ \texttt{option} = \texttt{None} \mid \texttt{Some}\ \alpha$$

Partial functions $\sigma \hookrightarrow \tau$ are now modeled using the total function $\sigma \Rightarrow \tau\ \texttt{option}$, where `None` values are used to represent undefined values, and $\texttt{Some}\ y$ is used to represent defined values y. We use this approach of simulating partial functions in HOL to represent the notion of object store (as a partial function from object identifiers to object values—see Section 5.2). Optional values are also used to keep track of modifications to variables (see Section 5.8).

- **Case** statements are used to perform case analysis, typically for data types such as lists and options. For example, `case` e `of None` $\Rightarrow 0 \mid$ `Some` $y \Rightarrow y + 1$ returns 0 if e is `None` and $y + 1$ for any `Some`-tagged value y. Unlike in ML, all cases have to be enumerated explicitly and in the same order as they occur in the corresponding data type definition.

4.2.3 Theorems, axioms and rules

Theorems are derived from axioms and/or previously proven theorems. Axioms and theorems take the same form and are often called *rules*. Inference rules take the form

$$[\!|\ A_1; \cdots; A_n\ |\!] \Longrightarrow A$$

to represent an implication with assumptions $A_1 \cdots A_n$ and conclusion A (the brackets are omitted if there is only one assumption). In HOL, the conclusion and the assumptions should both be terms of type `bool`[1]. For example, the following rules are axioms in HOL:

$$\begin{array}{rl} [\!|\ P \to Q; P\ |\!] \Longrightarrow Q & (\to E) \\ (P \Longrightarrow Q) \Longrightarrow P \to Q & (\to I) \end{array}$$

The first rule describes *what* to infer from $(P \to Q)$, while the second rules describes *how* to infer $(P \to Q)$. The first rule is called an *elimination rule*, while the second rule is called an *introduction rule*. Normally, elimination rules are used for reasoning forward from the assumptions, while introduction rules are for reasoning backwards from the conclusion (see Section 4.3 for an example).

The rule language also supports quantifiers; in this case, the rules can have both bound and free variables. The following rules provide the natural deduction style introduction and elimination properties of set-bounded universal quantification:

$$\begin{array}{rl} [\!|\ \forall x \in A \mid P\ x; P\ t \Longrightarrow Q; t \notin A \Longrightarrow Q\ |\!] \Longrightarrow Q & (\forall_\in E) \\ (!!x.x \in A \Longrightarrow P\ x) \Longrightarrow \forall x \in A \mid P\ x & (\forall_\in I) \end{array}$$

Both rules are derived theorems in HOL. The first rule eliminates the quantifier by introducing a new *unknown variable*, namely t. Typically, in reasoning using this rule t should be instantiated in a later stage of the proof (an example is shown in Section 4.3). In Isabelle, rules that introduce unknown variables are called *unsafe*, while rules that do not introduce such variables are called *safe*. The distinction between safe and unsafe is important, because it determines the order in which rules should be applied (safe rules should be applied before any unsafe rules are tried, see Section 4.3).

The second rule for introduction of the set-bounded universal quantification introduces a *parameter* or *eigenvariable* x, where !! is a construct of Isabelle's meta-logic to universally bind variables.

[1]In fact, they should be meta-logical propositions, but the encoding of HOL formulae in terms of meta-level propositions is invisible to the user.

Isabelle represents rules as ML values of type `thm`. Theorems are named using ML identifiers. For example, the rule $(\wedge I)$ mentioned above is identified by the ML identifier `impI`.

4.2.4 Tactics and tacticals

New theorems are proved mainly in a backward fashion. Backward reasoning is performed by refining the initial *proof goal* (i.e., the theorem we wish to prove) to progressively simpler subgoals until no subgoals remain.

Tactics Individual steps in a proof are made by applying *tactics*. Isabelle predefines a variety of tactics, including basic tactics for interactive proof as well as powerful tactics for automating proof steps:

- **Resolution** tries to unify the conclusion of a rule with a given subgoal, and on successful unification this goal is replaced by the instantiated assumption of the rule.

- **Proof by assumption** tries to unify the conclusion of a goal with one of the assumptions, and on successful unification deletes this goal.

- **Contradiction** tries to solve a goal by contradiction; the tactic succeeds if one of the goal's assumptions is the negation of another assumption.

- **Simplification** tactics are available through the *Simplifier* package. The Simplifier permits rewriting with an arbitrary set of rewrite rules. Rewrite rules are rules with a conclusion of the form $A = B$, where each occurrence of A in the goal is replaced by B. Isabelle installs over a few hundred standard rewriting rules for HOL, and new rules can be easily added. Many advanced features in term-rewriting are supported, such as the handling of conditional rewrite rules, the splitting of conditionals (if-then-else), and the use of congruence rule. We will highly benefit from these features for analyzing method code (a more detailed explanation follows in Section 7.2.2).

- **Classical reasoning** is supported through the *Classical Reasoner* package. The package defines several powerful tactics, for automated theorem proving in higher-order logic. These tactics handle substantial proofs, including quantifier reasoning. The tool uses a set of introduction and elimination rules for higher-order logic to automate natural deduction inferences. The default configuration of the tool includes machinery to reason about sets, lists, tuples, booleans, etc. The tool implements a depth-first search strategy; variables introduced by the use of quantifiers can automatically be instantiated, and backtracking is performed between different alternative unifiers. Between deduction steps, the Simplifier can be used to permit rewriting. This provides a powerful tool for automated reasoning in HOL. The combined Simplifier and Classical Reasoner will provide the basis for our automated analysis technique to reason about methods in object-oriented databases. An example proof that demonstrates the capabilities of the Classical Reasoner is discussed in Section 4.3.

- **Induction** is done by a special tactic. We do not address recursion and inductive proofs in this thesis. The database language we use has a limited expressiveness, going as far as Qian's set-bounded iteration [Qia91].

Tacticals Tactics provide implementations for individual proof steps. *Tacticals* are used to combine tactics, e.g. sequentially using tac_1 `THEN` tac_2 or tac_1 `ORELSE` tac_2 as alternatives.

4.3 A simple proof of an invariant using Isabelle

An example is shown of the proof of an invariant for a transaction on a relational database, to illustrate the use of tactics and basic quantifier reasoning in Isabelle.

We consider a simple relational database with a table R that has only one field x of type integer. There is a constraint that all elements of R should be smaller than 1600. Further, a simple transaction $T(R, a) \equiv R - \{a\}$ is defined, which deletes a given value a from R. Our goal is to show that the constraint is an invariant of the transaction. This amounts to proving the following theorem in HOL;

$$(\forall x \in R \mid x < 1600) \rightarrow (\forall x \in (R - \{a\}) \mid x < 1600)$$

That is, we assume that the constraint holds in a state R to show that it still holds after the application of T, for arbitrary R of type $(\texttt{int})\texttt{set}$ and a of type `int`. To start the proof of this theorem, we type the following command in Isabelle :

$>$ `Goal` "$(\forall x \in R \mid x < 1600) \rightarrow (\forall x \in (R - \{a\}) \mid x < 1600)$"

The above command asserts the required theorem as a *proof goal* (the symbol $>$ is the command prompt). Isabelle responds with the following initial goal:

`Level` 0
$(\forall x \in R \mid x < 1600) \rightarrow (\forall x \in (R - \{a\}) \mid x < 1600)$
 1. $(\forall x \in R \mid x < 1600) \rightarrow (\forall x \in (R - \{a\}) \mid x < 1600)$

In the goal, Isabelle takes the variables R and a to be implicitly universally quantified and automatically infers their types. The level number is an indication of the number of proof steps performed sofar. Initially this value is set to zero and it is increased each time a tactic is applied. Isabelle then shows the theorem that we try to prove, followed by the list of remaining *subgoals*. Initially there is only one subgoal, namely the theorem we wish to prove.

The proof is found automatically by Isabelle, using the depth-first search tactic of the Classical Reasoner (i.e., using `Fast_tac`). The proof takes almost zero seconds of proof time on an average workstation and involves basic quantifier reasoning. It is instructive to examine some details of the proof, interactively in Isabelle.

The interactive proof is performed in 6 steps. We first resolve the goal with the introduction rule for implication $(\wedge I)$, which is represented by the ML identifier `impI`.

```
> by (rtac impI 1);
Level 1
(∀x ∈ R | x < 1600) → (∀x ∈ (R − {a}) | x < 1600)
 1. (∀x ∈ R | x < 1600) ⟹ (∀x ∈ (R − {a}) | x < 1600)
```

The resolution tactic (`rtac`) is a function that takes the required theorem and subgoal; `by` is used to apply the tactic to the current goal. In this case, unification succeeds and Isabelle returns the modified proof state.

We may proceed in two directions: by reasoning forward from the assumptions using the rule $\forall_{\in}E$ to eliminate the quantifier, or by reasoning backwards from the conclusion using the rule $\forall_{\in}I$. The order in which we apply these rules is crucial in natural deduction: applying $\forall_{\in}E$ first leads to an unsuccessful proof, whereas $\forall_{\in}I$ succeeds. As a simple rule of thumb, we should first apply any safe rules (i.e., rules that do not introduce new unknown variables in the goal—see Section 4.2.3), before any unsafe rules are tried.

We resolve the conclusion of the goal with the rule labeled $(\forall_{\in}I)$. The same resolution tactic is used as in the previous step, but now applied to the rule `ballI` (i.e., the ML identifier that represents $(\forall_{\in}I)$).

```
> by (rtac ballI 1);
Level 2
(∀x ∈ R | x < 1600) → (∀x ∈ (R − {a}) | x < 1600)
 1. !!x.[| ∀x ∈ R | x < 1600 ; x ∈ (R − {a}) |] ⟹ x < 1600
```

We now eliminate the quantifier in the list of assumptions using the rule labeled $(\forall_{\in}E)$ (written as `ballE` in ML). Elimination rules are applied using `etac` instead of `rtac` (although this is just a combination of normal resolution and proof by assumption). Its effect with the rule $(\forall_{\in}E)$ applied to our goal is to create two subgoals and to introduce an unknown:

```
> by (etac ballE 1);
Level 3
(∀x ∈ R | x < 1600) → (∀x ∈ (R − {a}) | x < 1600)
 1. !!x.[| x ∈ (R − {a}) ; ?t x < 1600 |] ⟹ x < 1600
 2. !!x.[| x ∈ (R − {a}) ; ?t x ∉ R |] ⟹ x < 1600
```

The unknown variable $?t$ is a *function* unknown: on unification, Isabelle has lifted the unknown in $(\forall_{\in}E)$ over the parameter variable x in the goal. The function application $?t\ x$ may be replaced by any term containing x. The term we require is x itself, with $?t$ instantiated to the identity function $\lambda x \cdot x$. Notice that we would have got stuck at this point if $(\forall_{\in}E)$ and $(\forall_{\in}I)$ had been applied in opposite order: the unknown variable $?t$ would not be dependent on x. In our case, the instantiation of $?t\ x$ is found using proof by assumption on the first subgoal.

```
> by (assume_tac 1);
Level 4
(∀x ∈ R | x < 1600) → (∀x ∈ (R − {a}) | x < 1600)
 1. !!x.[| x ∈ (R − {a}) ; x ∉ R |] ⟹ x < 1600
```

The remaining subgoal is obviously true. It is solved systematically using the elimination rule for set difference $[| \ c \in A - B \ ; \ [| \ c \in A \ ; \ c \notin B \ |] \Longrightarrow P \ |] \Longrightarrow P$, which is part of the standard Isabelle/HOL library (and labeled `DiffE` in ML). Using `etac` with this rule, we obtain:

```
> by (etac DiffE 1);
Level 5
(∀x ∈ R | x < 1600) → (∀x ∈ (R − {a}) | x < 1600)
 1. !!x.[| x ∈ R ; x ∉ R ; x ∉ {a} |] ⟹ x < 1600
```

The remaining goal is solved using `contr_tac`, which detects the contradiction between the first and the second assumption:

```
> by (contr_tac 1);
Level 6
(∀x ∈ R | x < 1600) → (∀x ∈ (R − {a}) | x < 1600)
No subgoals!
```

4.4 Summary

- The type system used in HOL was designed to be similar to the functional programming language ML.
- Nested data structures, such as (`int` × `bool`) `set` are supported; these data structures are needed for object-oriented databases where similar data structures are found (see Chapter 3).
- Theorems are proved through the use of tactics; Isabelle has a suite of powerful tactics for automatically solving goals in higher-order logic. These tactics will provide the basis for our automated analysis of method code (see Chapter 7 and Chapter 8).
- An example proof has been shown, which demonstrates the capabilities of Isabelle's automated search tactics: a simple case of (relational) database transaction safety is verified in almost zero seconds of proof time on a normal workstation. In Chapter 7, we will consider similar proofs for the object-oriented case.

Chapter 5

Higher-order logic representation of an object-oriented database schema

A formal translation of an OASIS database schema to higher-order logic is defined. This translation provides the foundation for the practical verification work using Isabelle, discussed in the later chapters of this thesis. The translation to higher-order logic (HOL) implements a *semantic embedding* of the database language. The database state is translated to data structures in HOL; imperative methods are translated to functions that operate on the database state. The chapter gives details of how this is done.

5.1 Semantic embedding

Formal reasoning about programs of a language L relies on some mathematical foundation of L, usually *denotational semantics*. In denotational semantics, one defines semantic functions (e.g., $\mathcal{E}[\![-]\!]$), which map the syntactic constructs available within L to elements of some semantic domain D. Using this function, one can prove properties about a program p, by reasoning about its *denotation* $\mathcal{E}[\![p]\!]$ in D. To prove properties about programs using a higher-order logic theorem prover, a similar approach is taken [JvdBH+98].

There are two well-known approaches to formalize the semantics of a language in HOL: so-called *deep embedding* and *shallow embedding* [BGG+92]. A *deep embedding* means that the syntax of the source language, as well as semantics functions are encoded *within* HOL to assign meanings to programs. A *shallow embedding* means that the syntax and semantics functions are not part of the logical representation itself; rather, an external schema translator is used to parse the source syntax directly to semantic structures in HOL. In other words, a shallow embedding is one which identifies the constructs of the source language with similar constructs available in HOL. For example, the types of the source language are mapped to similar types in HOL. In this case, the schema translator "computes" the semantics equations.

The level of embedding is important. A deep embedding allows meta-level reasoning about the semantics equations (e.g, one could prove a general result of the form "for all programs"). Some general results are usually obtainable under a shallow embedding, by proving properties of semantic operators, but results of the form "for all programs" generally remain out of reach. A shallow embedding is typically more adequate if one wants to reason only about specific programs rather than strive for general results about the language itself.

The main advantage of a shallow embedding is that one can directly use the machinery of HOL to reason in the source language.

The present chapter defines a shallow embedding of OASIS in HOL. The choice for a shallow embedding follows naturally, because the goal of this work is to reason about specific method definitions, rather than proving general results.

The shallow embedding of OASIS in HOL makes use of the predefined Isabelle/HOL data types discussed in the previous chapter. These meet the demands of object-oriented database programming to a large extent; complex values are already supported by the system (e.g., a set of lists of integer values). In Section 5.5, we supplement the standard data types by a generic model of the memory (called object store). Specific methods are encoded in terms of this model. The encoding makes use of three semantics translation functions (as found in Appendix D):

$\mathcal{T} : typ \rightarrow holtyp$	(mapping of types, see Section 5.6)
$\mathcal{E} : exp \times env \rightarrow holexp$	(mapping of expressions, see Section 5.7)
$\mathcal{C} : com \times env \rightarrow holexp$	(mapping of commands, see Section 5.8)

These functions are defined at the *meta-level* (not as part of the logical representation in Isabelle), and intended for implementation in ML. The output is *syntax*: OASIS syntactic constructs are mapped to HOL syntax in a format that can be processed by the Isabelle/HOL theorem prover. The OASIS to HOL schema translation is, in some sense, a semantics mapping, where the output is HOL notation. The mapping is less rigorous than a denotational definition (i.e., [Win93]), in that the output notation (roughly corresponding to domains) is only intended to be reasoned about by the Isabelle system. Some aspects of the mapping are declarative in nature. For example, the "newness" of new oids only needs to be specified declaratively as an assumption in proofs; new oid values do not need to be computed.

The essential ingredients of the shallow embedding of OASIS in HOL (namely, a *generic* model of the object-store, a database specific model of the object store, and methods as functions on the store) are discussed below.

5.2 The generic object store

The object store is modeled as a partial function from object identifiers to values. In HOL, we represent such functions using the predefined '`option`' data type, as '`oid` $\Rightarrow \beta$ `option`'. HOL function types ($\Rightarrow$) are total, and partial function types are modeled using options. An option is a variant type, with two cases: `None` for the undefined cases, and `Some` y for a defined cases (see Section 4.2.2). The type variable β in the co-domain type will be instantiated differently for each schema, with a concrete type that describes the schema-specific class structures.

Based on this abstract notion of object store, a number of generic operations for retrieval and update are defined. Figure 5.1 gives signature declarations for these operations. The operations `oids`, `get`, and `eval` are used to retrieve information from the state. For example, the operation `get` is used for the translation of attribute selection. The other operations in the figure are used to update the state; they result in a "little" object store, which comprises local

$$
\boxed{
\begin{array}{l}
\texttt{oids} :: (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \Rightarrow \texttt{oid set} \\
\texttt{eval} :: [\beta\ \texttt{option}, \beta \Rightarrow \texttt{bool}] \Rightarrow \texttt{bool} \\
\texttt{get} :: [\beta\ \texttt{option}, \beta \Rightarrow \alpha] \Rightarrow \alpha \\
\texttt{set} :: [(\texttt{oid} \Rightarrow \beta\ \texttt{option}), \texttt{oid}, \beta \Rightarrow \beta] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{smash} :: [(\texttt{oid} \Rightarrow \beta\ \texttt{option}), (\texttt{oid} \Rightarrow \beta\ \texttt{option})] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{apply} :: [\alpha\ \texttt{set}, [\alpha, \texttt{oid}] \Rightarrow \beta\ \texttt{option}] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{new} :: [\texttt{oid}, \beta] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\
\texttt{skip} :: (\texttt{oid} \Rightarrow \beta\ \texttt{option})
\end{array}
}
$$

Figure 5.1: Generic operations for objects

changes to the state. For example, the operation `set` is used for the translation of attribute assignment. The operation `smash` is used for sequential composition (‘`;`’). Details about the definitions of these operations and useful theorems are given in Section 5.5.

5.3 The database-specific object store: class declarations as data types

The type variable β that appears in the generic object store type is instantiated with a type `object`. This type is different for each schema; it describes the database specific class structures defined in the schema. To describe the domain of object values for a given schema, we use a variant type.[1] Cases are introduced for each of the concrete classes in the database schema. Abstract classes are omitted because such classes cannot be instantiated (they are merely used for type generalization). For the case study example (see Appendix A), the following `object` type is obtained:

```
datatype object = AtomicContents string string string
                | ANode string int (oid set) (oid set) (oid set)
                | CNode string int (oid set) (oid set) int (oid set)
                | Link string int oid oid
```

Structural information for an object (attribute values) is supplied as an argument to its data type constructor. This information includes all attributes inherited from super-classes. For example, instantiations of the class ANode have five attributes: one attribute name of type string inherited from Element, the attribute position of type int, incomingLinks and outgoingLinks both of type set⟨Link⟩, inherited from Node, and the attribute content of type set⟨string⟩ defined in ANode. In the definition of the data type, only the corresponding HOL types of the attributes are listed. These types are similar to the OASIS types, but there is an important difference. Class references in compound objects appear as “pointer” references

[1]This is analogous to the use of the disjoint union domain in denotational semantics, as a means to put values from different domains in the same set [MS76].

in the form of oid-values. For example, the type set⟨Link⟩ is mapped to the type $(\texttt{oid set})$. Thus heterogeneous collections of objects map to homogeneous collections of oids in HOL.

Mutual recursive classes do not present a problem in our representation, because objects are handled as references. For example, the classes Node and Link in the case study example are mutual recursive (Link refers to Node and vice versa); the use of oids "flattens" the recursive type structures. Further, the constructors of type `object` provide *run-time* type information, which is needed to cover *late binding*.

5.4 Methods as functions on the database state

Methods are represented as functions in HOL. A *retrieval method* maps an input object store, persistent roots, an oid for the receiver object, and actual parameter values to a (complex) value that represents the return value of the method. For the encoding of the method body, we use the operations from Figure 5.1, as well as *predefined* Isabelle/HOL constructs. These include the standard operations on set, lists, integers, boolean, etc. The additional operations `eval` and `get` are used to encode operations on objects. As an example, we look at the use of `get` to encode attribute selections, in the HOL definition of the method isConnectedTo of the example schema. This method is an abstract method of class Element, with different implementations for class Node and Link (see Section 3.1.3). Here is the HOL code for the implementation in class Node:

$$\begin{array}{l}
\texttt{Node\_isConnectedTo} :: [\texttt{OS}, \overbrace{(\texttt{oid set}), (\texttt{oid set}), (\texttt{oid set})}^{\text{persistent roots}}, \texttt{oid}, \texttt{oid}] \Rightarrow \texttt{bool} \\
\texttt{Node\_isConnectedTo os cnodes links anodes this n} \equiv \\
\qquad \texttt{n} \in (\texttt{get (os this)}\ \mathit{inLinksOf}) \vee \texttt{n} \in (\texttt{get (os this)}\ \mathit{outLinksOf})
\end{array}$$

The method is defined as a function acting on object store, persistent roots, this, and parameter. The name of the method is prefixed with the name of the defining class. This prevents name clashes due to overloading. In the signature, '`OS`' abbreviates the database specific object store type '$\texttt{oid} \Rightarrow \texttt{object option}$'. The other parameters are for the persistent roots, the receiver object and the method parameter. The `get`-operation is used to look-up the values of incomingLinks and outgoingLinks-attributes of the receiver object; the free variables '$\mathit{inLinksOf}$' and '$\mathit{outLinksOf}$ are abbreviations of functions that encode the actual attribute selections. Here is the definition of the function $\mathit{inLinksOf}$:

$$\begin{array}{l}
\mathit{inLinksOf} \equiv \\
\quad \lambda\texttt{val} \cdot \texttt{case val of AtomicContents refDir showStat URL} \Rightarrow \texttt{arbitrary} \\
\qquad\qquad\qquad\qquad \mid \texttt{ANode name pos ins outs content} \Rightarrow \texttt{ins} \\
\qquad\qquad\qquad\qquad \mid \texttt{CNode name pos ins outs size elements} \Rightarrow \texttt{ins} \\
\qquad\qquad\qquad\qquad \mid \texttt{Link name pos from to} \Rightarrow \texttt{arbitrary}
\end{array}$$

The `get` application returns the value of the incomingLinks-attribute if '`n`' is `Some`-tagged in the object store, with the right type (Node); otherwise an *arbitrary* value is returned, which represents an undefined function result in HOL (see Section 4.2.2).

The above example showed how read-only methods are represented in HOL, using the abstract retrieval operations from Figure 5.1. Methods with side effects are more complicated. In this case, the output is a *tuple* that includes a component for the return value of the method, and an additional component for the modifications to the object store. It also includes components for modifications to the persistent roots and the method parameters, which we ignore in the presentation (see Section 5.8 for details).

The operations `skip`, `set`, `new`, `smash` and `apply` are used to encode updates to the object store. As an example, we look at the use of `set` to encode attribute updates, in the HOL definition of the method removeElement method:

$\texttt{CNode\_removeElement} :: [\texttt{OS}, \texttt{oid set}, \texttt{oid set}, \texttt{oid set}, \texttt{oid}, \texttt{oid}] \Rightarrow (\texttt{OS} \times \texttt{bool})$
$\texttt{CNode\_removeElement os cnodes links anodes this n} \equiv$
$\quad \texttt{if } \mathit{cond} \texttt{ then } (\texttt{set os this } \mathit{delElt}, \texttt{True}) \texttt{ else } (\texttt{skip}, \texttt{False})$

The first component of the result tuple is a Δ-value [DHDD95]. It corresponds to a “little” object store, which only contains bindings for the modified objects. In this example, only the receiver object is modified. The modification is encoded using ‘`set`’: the function argument f encodes the modification to the elements field of the receiver object, by removing oid ‘`n`’ from the set. Function $delElt$ is defined as follows:

```
delElt ≡
    λ val · case val of AtomicContents refDir showStmt URL ⇒
                        AtomicContents refDir showStmt URL
                       | ANode name pos ins outs content ⇒
                        ANode name pos ins outs content
                       | CNode name pos ins outs size elts ⇒
                        CNode name pos ins outs size (elts − {n})
                       | Link name pos from to ⇒ Link name pos from to
```

It maps an `object` value to an `object` value. Observe that `n` is a free variable; it is bound by the arguments to the function `CNode_removeElement`. More complex update expressions are constructed by referring to e.g., the (input) object store and persistent roots.

The condition ‘$cond$] of the method involves *late binding*. It is represented using a number of applications of the ‘`eval`’ and ‘`get`’ operations of the theory of objects:

```
cond ≡
 (eval (os n) isElt ∧ n ∈ (get (os this) eltsOf))∧
      (∀x ∈ (get (os this) eltsOf) |
         ¬ (if (eval (os x) isLink)
            then (Link_isConnectedTo os cnodes links anodes x n)
            else (if (eval (os x) isNode)
                  then (Node_isConnectedTo os cnodes links anodes x n)
                  else arbitrary)))
```

The first occurrence of `eval` represents the ‘n != nil’ comparison of the removeElement method. The `eval` operation is applied to the storage cell associated with `n`, and the function $isElt$:

isElt ≡
```
  λ val · case val of AtomicContents refDir showStmt URL ⇒ False
                    | ANode name pos ins outs content ⇒ True
                    | CNode name pos ins outs size elts ⇒ True
                    | Link name pos from to ⇒ True
```

The above function returns `True` if `n` is `Some`-tagged in the object store, with the right type (see Section 5.7.1 for details).

The second subexpression in the definition of $cond$ checks that `n` is in the '`elements`' field of the '`this`' object. The function $eltsOf$ encodes the selection of the elements attribute (details of the encoding of attributes are discussed in Section 5.7.2):

eltsOf ≡
```
λval · case val of AtomicContents refDir showStat URL ⇒ arbitrary
                 | ANode name pos ins outs content ⇒ arbitrary
                 | CNode name pos ins outs size elements ⇒ elements
                 | Link name pos from to ⇒ arbitrary
```

The last subexpression is the HOL representation of the following test in the method definition: forall x in elements : not(x.isConnectedTo(n))). This involves *late binding*: based on the (run-time) type of `x`, the appropriate version of the isConnectedTo method is applied; an '`arbitrary`' value results for a wrongly typed or nil `x`.

Object creation is one of the complicating factors in reasoning about object-oriented software. It involves the implicit generation of new oids, which must be made explicit in the formal representation. The schema translator takes care of oid generation: it augments the function representation of a method with a sufficient number of extra parameters, which supply the new oid values. For example, the representation of a method creating just one new object is augmented with an extra parameter `nid` of type `oid`. More complex examples are given in Section 5.8.

The rest of this chapter shows details of the OASIS/HOL embedding. It documents the algorithms that have been implemented in ML, to translate OASIS database schemas to HOL.

5.5 A simple theory of objects in HOL

In this section, we define a HOL theory of object-oriented systems. This theory introduces several higher-order functions for database retrieval and update, based on an abstract notion of database state. These functions are modeled as schema independent operations, which take functions as parameters to make them specific. The following sections discuss the definition of the operations shown in Figure 5.1 and gives theorems derived from these definitions using the Isabelle system.

5.5.1 The database state

The database state (also called the *object store*) is modeled as a partial function from object identifiers to values. In Isabelle/HOL we represent such functions using the predefined ‘`option`’ data type, as follows:

$\texttt{oid} \Rightarrow \beta\ \texttt{option}$ (object store type)

HOL functions are total, but partial functions can be simulated using options. The option data type

$\texttt{datatype}\ \alpha\ \texttt{option} = \texttt{None} \mid \texttt{Some}\ \alpha$

that comes with the standard distribution of Isabelle/HOL includes the constructor `None` which is used to represent *undefined* function results, and the constructor `Some` used for *defined* function results (in this case, the actual value is supplied as an argument to `Some`). The type variable β in the co-domain type can be instantiated with an arbitrary object type that captures the database-specific class information. Hence, the above type declaration serves as a template for actual database structures. The type of object identifiers (`oid`) is declared a synonym of the the type `nat option`, as follows:

$\texttt{types oid} = \texttt{nat option}$ (oid type)

Informally, oids can be `nil` (represented as a `None` value), or allocated (`Some`-tagged).

5.5.2 Retrieval operations

Once we have defined the database state, several abstract retrieval operations can be defined to extract information from the state. These are schema-independent operations that can be applied to arbitrary database structures. We define a function `oids` that extracts the domain of a given object store, as follows:

$\texttt{oids} :: (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \Rightarrow \texttt{oid set}$
$\texttt{oids}\ os \equiv \{x \mid (os\ x) \neq \texttt{None}\}$

Note the declarative nature of the definition. Isabelle does not actually compute such comprehensions (it only provides a way to reason about them).

A predicate ‘`eval`’ is defined to validate whether a given object ‘$\beta\ option$’ value satisfies some predicate p.

$\texttt{eval} :: [\beta\ \texttt{option}, \beta \Rightarrow \texttt{bool}] \Rightarrow \texttt{bool}$
$\texttt{eval}\ obj\ p \equiv \texttt{case}\ obj\ \texttt{of None} \Rightarrow \texttt{False} \mid \texttt{Some}\ z \Rightarrow p\ z$

Typically, predicate p is a filter expression that reflects database-specific typing constraints. The definition of `eval` uses a case-split on ‘obj’: if its value is undefined, `False` is returned; for defined objects, the predicate p is applied to the object’s actual value component z.

To select information from the value component of an object, given its oid, we introduce a generic function ‘`get`’:

$\texttt{get} :: [\beta\ \texttt{option}, \beta \Rightarrow \alpha] \Rightarrow \alpha$
$\texttt{get}\ obj\ f \equiv \texttt{case}\ obj\ \texttt{of None} \Rightarrow \texttt{arbitrary} \mid \texttt{Some}\ z \Rightarrow f\ z$

The definition of `get` is motivated by its intended use in attribute selection. The application '`get` $obj\ f$' applies the function f to the object value 'obj'. In case the value is undefined, an *arbitrary* value is returned (written `arbitrary`). Arbitrary values are available for all domains and are part of the standard Isabelle/HOL theory (see Chapter 4). Here it takes type α.

5.5.3 Update operations

We define functions that correspond to the update constructs of the OASIS database specification language. These functions are modeled as schema independent operations, which take (functions as) parameters to make them specific. Formally, they are represented as mappings from database states to *deltas* in the approach of [DHDD95]. A delta might be conceived of as a 'little' database state, which only contains bindings for modified objects. Hence, deltas are modeled using the same type as the object store structure. Keeping the modified objects separate from the ones that are not updated, allows us to define several combinators for merging partial results.

To modify the value component of an object, given its oid, we introduce a generic function `set`, which complements the function `get`:

$$\begin{array}{l}\texttt{set} :: [(\texttt{oid} \Rightarrow \beta\ \texttt{option}), \texttt{oid}, \beta \Rightarrow \beta] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option})\\ \texttt{set}\ os\ y\ f \equiv \lambda x \cdot \texttt{if}\ (x = y)\ \texttt{then}\ (\texttt{case}\ (os\ y)\ \texttt{of None} \Rightarrow \texttt{None} \mid \texttt{Some}\ z \Rightarrow \texttt{Some}\ (f\ z))\\ \qquad\qquad\qquad\quad \texttt{else None}\end{array}$$

The application '`set` $os\ ida\ f$' returns a function The function takes an oid value 'x', and if it is equal to 'id' it returns the corresponding modified value representation; all other oid values are mapped to None. The function returned by an application of the `set` construct corresponds to a *delta value*, which only contains a binding for the modified object (ida).

New object representations are constructed using the function `new`, which we define as follows:

$$\begin{array}{l}\texttt{new} :: [\texttt{oid}, \beta\ \texttt{option}] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option})\\ \texttt{new}\ y\ val \equiv \lambda x \cdot \texttt{if}\ (x = y)\ \texttt{then}\ val\ \texttt{else None}\end{array}$$

Similar to applications of `set`, the `new` function returns the new object representation as a function with a single defined value. New oid generation is left to the calling environment. The mapping of OASIS commands to HOL introduces extra parameter variables for invocations of `new`. These parameter variables have to be supplied appropriate values in proofs. This is discussed extensively in Section 5.8.4. Observe that, since persistence by reachability is used, there is no need to include the complementary function to delete an object from the store.

An identity constant `skip` is defined, which corresponds to the OASIS skip operation, which performs no actions at all:

$$\begin{array}{l}\texttt{skip} :: \texttt{oid} \Rightarrow \beta\ \texttt{option}\\ \texttt{skip} \equiv \lambda x \cdot \texttt{None}\end{array}$$

We define several operations to combine delta and/or object store values. These operations correspond to the control structures in the update language. Following [DHDD95], we define an operator for combining deltas, called *smash*:

$$\begin{array}{l}\texttt{smash} :: [(\texttt{oid} \Rightarrow \beta\ \texttt{option}), (\texttt{oid} \Rightarrow \beta\ \texttt{option})] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option}) \\ \texttt{smash}\ \Delta_1\ \Delta_2 \equiv \lambda x \cdot \texttt{case}\ (\Delta_2\ x)\ \texttt{of None} \Rightarrow (\Delta_1\ x) \mid \texttt{Some}\ y \Rightarrow \texttt{Some}\ y\end{array}$$

The `smash` operator loosely relates to the sequential composition operator found in imperative programming languages. It is used to represent sequential composition in the OASIS update language.

For the representation of the if-then-else operator, we do not need to define an additional construct. We use the predefined Isabelle/HOL conditional branch operator.

We define an additional operator called *apply*, which will serve as a basis for the bounded-iteration construct. The operation is defined as follows:

$$\begin{array}{ll}\multicolumn{2}{l}{\texttt{apply} :: [\alpha\ \texttt{set}, [\alpha, \texttt{oid}] \Rightarrow \beta\ \texttt{option}] \Rightarrow (\texttt{oid} \Rightarrow \beta\ \texttt{option})} \\ \texttt{apply}\ A\ f \equiv \lambda x \cdot & \texttt{if}\ (\exists! y \in A \mid x \in (\texttt{oids}(f\ y))) \\ & \texttt{then}\ (\epsilon\ z \cdot \forall y \in A \mid x \in \texttt{oids}(f\ y) \rightarrow z = f\ y\ x) \\ & \texttt{else None}\end{array}$$

The `apply` operation takes two arguments: the first is a set which is iterated over, and the second is a delta function that depends on the iteration variable. The application '`apply` $A\ f$' constructs a new delta, that accumulates the effects of the deltas that result when 'f' is applied to the elements of 'A'. Ideally, the resulting deltas do not designate the same objects, but because of sharing conflicts of multiple incompatible updates to one and the same object might arise. The definition of `apply` deals with such conflicts. Given an oid 'x', the new delta checks whether there is a modification to 'x' for exactly one of the elements of A. In this case, that unique modified value 'z' is selected using Hilbert's ϵ-operator; otherwise, `None` is returned. Hence, in the case that for different elements of 'A' an update is applied to the same 'x', those (conflicting) updates are ignored. This corresponds to a no-op in the case of conflicting set-oriented updates.

5.5.4 Theorems about the operations

Using Isabelle, we have derived a number of useful theorems about the retrieval and update operations discussed above. Equality theorems of the form $H \rightarrow LHS = RHS$ are derived, which are intended for term-rewriting purposes, using Isabelle's Simplifier tool. Introduction and elimination rules are given for `oids` and `eval`; these rules are intended for use with Isabelle's Classical Reasoner tool. The practical use of these rules and tools is discussed in Chapter 7 and Chapter 8.

Figure 5.2 lists some equality theorems we derived for Δ-pattern. This includes standard theorems for `smash` (such as associativity, reflexivity and left and right identity). Most likely, this set of rewrite rules can be further extended, and an extensive study for identifying recurring delta-patterns is left for future work.

Figure 5.3 lists equality theorems for the `oids` function, when applied to a delta argument. A named equation exists for all possible delta patterns generated for our input update

[smash_assoc]	$\texttt{smash}\ os_1\ (\texttt{smash}\ os_2\ os_3) = \texttt{smash}\ (\texttt{smash}\ os_1\ os_2)\ os3$
[smash_idemp]	$\texttt{smash}\ os\ os = os$
[smash_right_id]	$\texttt{smash}\ os\ \texttt{skip} = os$
[smash_left_id]	$\texttt{smash}\ \texttt{skip}\ os = os$
[smash_set_set]	$\texttt{smash}\ (\texttt{set}\ os\ x\ f)\ (\texttt{set}\ os\ x\ g) = \texttt{set}\ os\ x\ (g \circ f)$
[set_new]	$\texttt{set}\ (\texttt{smash}\ os\ (\texttt{new}\ x\ val))\ x\ f = \texttt{new}\ x\ (f\ val)$
[set_set]	$\texttt{set}\ (\texttt{smash}\ os\ (\texttt{set}\ os\ x\ f))\ x\ g = \texttt{set}\ os\ x\ (g \circ f)$

Figure 5.2: Rewrite rules for deltas

[oids_skip]	$\texttt{oids}(\texttt{skip}) = \{\}$
[oids_set]	$\texttt{oids}(\texttt{set}\ os\ ida\ f) = \{x \mid x \in \texttt{oids}\ os \wedge x = ida\}$
[oids_new]	$\texttt{oids}(\texttt{new}\ ida\ vala) = \{ida\}$
[oids_smash]	$\texttt{oids}(\texttt{smash}\ os_1\ os_2) = (\texttt{oids}\ os_2) \cup (\texttt{oids}\ os_1)$
[oids_apply]	$\texttt{oids}(\texttt{apply}\ A\ f) = \{x \mid \exists! y \in A \mid x \in \texttt{oids}(f\ y)\}$

Figure 5.3: Equality theorems for oids

language. In Figure 5.4 and Figure 5.5 equality theorems are given for applications of the `eval` and `get` function, respectively. These rules describe how to push the retrieval operation `get` and `eval` through a modified object store. Observe that, the extra smash in these rules should be read as an apply of the partial results in the delta to the object store (in this case the first argument of `smash` is an object store, not just a delta).

Introduction and elimination rules are derived for `oids` and `eval` (see Figures 5.6 and 5.7). These rules are intended for use with Isabelle's Classical Reasoner tool. The rules of `eval` are database-specific; the rules in Figure 5.7 are specific for the example database schema (see Apppendix A). These rules provide a form of case-based reasoning, which is discussed in Chapter 7 and Chapter 8.

5.6 Mapping of the object definition language

The schema is mapped to a HOL theory. This theory provides the definition of the database-specific object store, and definitions for the methods and constraints defined in the schema, as discussed in the previous sections. Part of the translation of the schema concerns the mapping of types, which is discussed below.

[eval_set] $\texttt{eval}\ ((\texttt{smash}\ os_1\ (\texttt{set}\ os_2\ idb\ f))\ ida)\ p =$
$\quad\texttt{if}\ (idb = ida) \wedge idb \in \texttt{oids}\ os_2\ \texttt{then}\ \texttt{eval}\ (os_2\ ida)\ (p \circ f)$
$\quad\quad\quad\texttt{else}\ \texttt{eval}\ (os_1\ ida)\ p)$

[eval_new] $idb \notin \texttt{oids}\ os \rightarrow$
$\texttt{eval}\ ((\texttt{smash}\ os\ (\texttt{new}\ idb\ valb))\ ida)\ p =$
$(idb = ida \wedge p(valb)) \vee (\texttt{eval}\ (os\ ida)\ p)$

[eval_smash] $\texttt{eval}\ ((\texttt{smash}\ os\ (\texttt{smash}\ os_1\ os_2))\ ida)\ p =$
$\quad\texttt{if}\ (ida \in \texttt{oids}\ os_2)\ \texttt{then}\ \texttt{eval}\ ((\texttt{smash}\ os\ os_2)\ ida)\ p$
$\quad\texttt{else}\ \texttt{if}\ (ida \in \texttt{oids}\ os_1)\ \texttt{then}\ \texttt{eval}\ ((\texttt{smash}\ os\ os_1)\ ida)\ p$
$\quad\quad\quad\texttt{else}\ \texttt{eval}\ (os\ ida)\ p$

[eval_apply] $\texttt{eval}\ ((\texttt{smash}\ os\ (\texttt{apply}\ A\ f))\ ida)\ p =$
$\quad\texttt{if}\ (\exists! x \in A \mid ida \in \texttt{oids}(f\ x)$
$\quad\texttt{then}\ (\exists x \in A \mid ida \in (\texttt{oids}(f\ x)) \wedge \texttt{eval}\ ((\texttt{smash}\ os\ (f\ x))\ ida)\ p)$
$\quad\texttt{else}\ \texttt{eval}\ (os\ ida)\ p)$

[eval_Some] $\texttt{eval}\ (\texttt{Some}\ y)\ p = p(y)$

Figure 5.4: Equality theorems for eval

[get_set] $\texttt{get}\ ((\texttt{smash}\ os_1\ (\texttt{set}\ os_2\ idb\ f))\ ida)\ g =$
$\quad\texttt{if}\ (idb = ida) \wedge idb \in \texttt{oids}\ os_2\ \texttt{then}\ \texttt{get}\ (os_2\ ida)\ (g \circ f)$
$\quad\quad\quad\texttt{else}\ \texttt{get}\ (os_1\ ida)\ g)$

[get_new] $\texttt{get}\ ((\texttt{smash}\ os\ (\texttt{new}\ idb\ valb))\ ida)\ g =$
$\quad\texttt{if}\ (idb = ida)\ \texttt{then}\ \{g(valb)\}\ \texttt{else}\ \texttt{get}\ (os\ ida)\ g$

[get_smash] $\texttt{get}\ ((\texttt{smash}\ os\ (\texttt{smash}\ os_1\ os_2))\ ida)\ f =$
$\quad\texttt{if}\ (ida \in \texttt{oids}\ os_2)\ \texttt{then}\ \texttt{get}\ ((\texttt{smash}\ os\ os_2)\ ida)\ f$
$\quad\texttt{else}\ \texttt{if}\ (ida \in \texttt{oids}\ os_1)\ \texttt{then}\ \texttt{get}\ ((\texttt{smash}\ os\ os_1)\ ida)\ f$
$\quad\quad\quad\texttt{else}\ \texttt{get}\ (os\ ida)\ f$

[get_apply] $\texttt{get}\ ((\texttt{smash}\ os\ (\texttt{apply}\ A\ f))\ ida)\ f =$
$\quad\texttt{if}\ (\exists! y \in A \mid ida \in \texttt{oids}(f\ y))$
$\quad\texttt{then}\ \epsilon\ z \cdot \forall y \in A \mid ida \in \texttt{oids}(f\ y) \rightarrow z = \texttt{get}\ ((\texttt{smash}\ os\ (f\ y))\ ida)\ g$
$\quad\texttt{else}\ \texttt{get}\ (os\ ida)\ f$

[get_Some] $\texttt{get}\ (\texttt{Some}\ y)\ f = f\ y$

Figure 5.5: Equality theorems for get

[oidsE] $[|\ x \in \texttt{oids}(os); os\ x \neq \texttt{None} \Longrightarrow R\ |] \Longrightarrow R$

[oidsI] $os\ x \neq \texttt{None} \Longrightarrow x \in \texttt{oids}(os)$

Figure 5.6: Introduction and elimination rule of `oids`

[evalE]

$$\begin{array}{l}
[\![\ \texttt{eval}\ o\ (\lambda z \cdot \texttt{case}\ z\ \texttt{of}\ \texttt{AtomicContents}\ \texttt{refDir}\ \texttt{showStat}\ \texttt{URL} \Rightarrow p_1 \\
\qquad\qquad\qquad\qquad \mid \texttt{ANode}\ \texttt{name}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{content} \Rightarrow p_2 \\
\qquad\qquad\qquad\qquad \mid \texttt{CNode}\ \texttt{name}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{size}\ \texttt{elts} \Rightarrow p_3 \\
\qquad\qquad\qquad\qquad \mid \texttt{Link}\ \texttt{name}\ \texttt{pos}\ \texttt{from}\ \texttt{to} \Rightarrow p_4); \\
\quad !!\ \texttt{ref}\ \texttt{shstat}\ \texttt{url}.[\!|\ o = \texttt{AtomicContents}\ \texttt{ref}\ \texttt{shstat}\ \texttt{url}; p_1\ |\!] \Rightarrow R; \\
\quad !!\ \texttt{n}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{cont}.[\!|\ o = \texttt{ANode}\ \texttt{n}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{cont}; p_2\ |\!] \Rightarrow R; \\
\quad !!\ \texttt{n}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{s}\ \texttt{elts}.[\!|\ o = \texttt{CNode}\ \texttt{n}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{s}\ \texttt{elts}; p_3\ |\!] \Rightarrow R; \\
\quad !!\ \texttt{n}\ \texttt{pos}\ \texttt{from}\ \texttt{to}.[\!|\ o = \texttt{Link}\ \texttt{n}\ \texttt{pos}\ \texttt{from}\ \texttt{to}; p_4\ |\!] \Rightarrow R\ |\!] \Rightarrow R
\end{array}$$

[evalI]

$$\begin{array}{l}
([\!|\ o \neq \texttt{AtomicContents}\ \texttt{refDir}\ \texttt{showStat}\ \texttt{URL} \vee \neg p_1; \\
\quad o \neq \texttt{ANode}\ \texttt{name}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{content} \vee \neg p_2; \\
\quad o \neq \texttt{CNode}\ \texttt{name}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{size}\ \texttt{elts} \vee \neg p_3; \\
|\!] \Longrightarrow o = \texttt{Link}\ \texttt{name}\ \texttt{pos}\ \texttt{from}\ \texttt{to} \wedge p_4) \Longrightarrow \\
\quad \texttt{eval}\ o\ (\lambda z \cdot \texttt{case}\ z\ \texttt{of}\ \texttt{AtomicContents}\ \texttt{refDir}\ \texttt{showStat}\ \texttt{URL} \Rightarrow p_1 \\
\qquad\qquad\qquad\qquad \mid \texttt{ANode}\ \texttt{name}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{content} \Rightarrow p_2 \\
\qquad\qquad\qquad\qquad \mid \texttt{CNode}\ \texttt{name}\ \texttt{pos}\ \texttt{ins}\ \texttt{outs}\ \texttt{size}\ \texttt{elts} \Rightarrow p_3 \\
\qquad\qquad\qquad\qquad \mid \texttt{Link}\ \texttt{name}\ \texttt{pos}\ \texttt{from}\ \texttt{to} \Rightarrow p_4);
\end{array}$$

Figure 5.7: Database-specific introduction and elimination rule of `eval`

5.6.1 Types and environments

The predefined HOL types `set`, `list`, `char`, `string`, `bool` and `int` can be directly used to represent similar OASIS types. At present, OASIS does not support bags (or multisets), since the standard distribution of HOL only includes a preliminary definition which is not sufficiently developed yet. The void type is used in typing commands that do not return a value; this type is mapped to the type `unit` (with the unique constructor '( )') in Isabelle/HOL.

Simple (unlabeled) tuple types are used to represent struct types, This is done in the usual way, by assuming an (alphabetic) ordering of the labels. In fact, this only yields an approximation of the record semantics, for it ignores the fact that the attributes contribute to record identity. For example, a record struct(a:10,b:20) is not the same as struct(b:10,c:20), while this can be derived in our HOL representation. Recently, Isabelle/HOL has been extended with labeled record types [NW98], which would solve these problems. Operations to access the fields of a record value are described in Section 5.7.

The meta type of environments env abbreviates the meta type $(string \times typ)list$. Environments are used to keep track of variable bindings (persistent roots, receiver, method parameters, and local variables). The environment $A = [(x_1, t_1), \cdots, (x_n, t_n)]$ represents an environment with n variable names $x_1 \cdots x_n$ and their associated types $t_1 \cdots t_n$. A new variable x of type t is added at the front of A, using the notations $(x, t) :: A$. The order in which variables are placed in the environment list is important: variable lookup is from head to tail. The initial environment of a method consists of (in the order listed) the methods parameters, the receiver "this", and the persistent roots. Local variable declerations are added at the front. An example is discussed below.

$$\begin{array}{ll}
& \mathcal{E}[\![\textsf{forall x in elements : not(x.isConnectedTo(n))}]\!](A) \\
= & \forall \mathtt{x} \in (\mathcal{E}[\![\textsf{elements}]\!](A)) \mid \mathcal{E}[\![\textsf{not(x.isConnectedTo(n))}]\!]((\textsf{x}, \textsf{Element}) :: A) \\
= & \forall \mathtt{x} \in (\mathtt{get}\ (\mathtt{os}\ \mathtt{this})\ eltsOf) \mid \mathcal{E}[\![\textsf{not(x.isConnectedTo(n))}]\!]((\textsf{x}, \textsf{Element}) :: A) \\
= & \forall \mathtt{x} \in (\mathtt{get}\ (\mathtt{os}\ \mathtt{this})\ eltsOf) \mid \neg(\mathcal{E}[\![\textsf{x.isConnectedTo(n)}]\!]((\textsf{x}, \textsf{Element}) :: A) \\
= & \forall \mathtt{x} \in (\mathtt{get}\ (\mathtt{os}\ \mathtt{this})\ eltsOf) \mid \\
& \qquad\quad \neg(\mathtt{if}\ (\mathtt{eval}\ (\mathtt{os}\ \mathtt{x})\ isLink) \\
& \qquad\qquad \mathtt{then}\ (\mathtt{Link\_isConnectedTo}\ \mathtt{os}\ \mathtt{cnodes}\ \mathtt{links}\ \mathtt{anodes}\ \mathtt{x}\ \mathtt{n}) \\
& \qquad\qquad \mathtt{else}\ (\mathtt{if}\ (\mathtt{eval}\ (\mathtt{os}\ \mathtt{x})\ isNode) \\
& \qquad\qquad\qquad \mathtt{then}\ (\mathtt{Node\_isConnectedTo}\ \mathtt{os}\ \mathtt{cnodes}\ \mathtt{links}\ \mathtt{anodes}\ \mathtt{x}\ \mathtt{n}) \\
& \qquad\qquad\qquad \mathtt{else}\ \mathtt{arbitrary})))
\end{array}$$

Figure 5.8: Example of mapping OQL expressions to HOL

5.7 Mapping of the object query language

Both OASIS-OQL and HOL are functional languages, and they share many commonly used language constructs and data types. As a consequence, many OQL expressions have a direct counterpart in the HOL syntax. In the informal introduction of this chapter, we already demonstrated the use of the generic operations `eval` and `get` to encode typical operations on objects, such as attribute selections and late binding. In this section, we highlight some other details of the mapping of expressions. A complete definition of the translation function $\mathcal{E}$ is found in the Appendix.

When applied to an expression e of type t, and environment $A = [(x_1, t_1), \cdots, (x_n, t_n)]$, the function $\mathcal{E}$ generates HOL output of the following type:

$$\lambda \mathtt{os} : \mathtt{OS} \lambda x_n : \mathcal{T}[\![t_n]\!] \cdots \lambda x_1 : \mathcal{T}[\![t_1]\!] \cdot (\mathcal{E}[\![e]\!]_A : \mathcal{T}[\![t]\!])$$

Observe that, $\mathcal{E}$ does not generate a closed λ-term: the object store and variables are captured by the surrounding environment (c.f., the definition of retrieval methods and constraints, in the previous section). Also observe that the output of $\mathcal{E}$ is not an optional value: exceptional behavior is dealt with using `arbitrary` values in HOL (see Section 61).

Figure 5.8 shows an example of how OQL expressions are mapped to HOL using the function $\mathcal{E}$. The example considers the translation of the following subexpression

$$\textsf{forall x in elements : not(x.isConnectedTo(n))}$$

in the condition of the removeElement method (see Appendix A). On page 51, we gave the HOL code that is generated for this expression; Figure 5.8, shows how this output is obtained by a the syntax directed translation using $\mathcal{E}$. The initial environment A of the method takes the persistent roots, the receiver, and the method parameter:

$$\begin{array}{l}
A = [(\textsf{n}, \textsf{Element}), (\textsf{this}, \textsf{CNode}), (\textsf{cnodes}, \textsf{set}\langle\textsf{CNode}\rangle), \\
\qquad (\textsf{links}, \textsf{set}\langle\textsf{Link}\rangle)(\textsf{anodes}, \textsf{set}\langle\textsf{ANode}\rangle)]
\end{array}$$

5.7.1 Literals, Nil values and Class Membership

The boolean constants, true and false, integer constants, enumerated sets, and list, are literals that can be represented straightforwardly using predefined constants of the Isabelle/HOL data type theories. The representation of the object literal 'nil' is more difficult. We distinguish two cases. Nil comparisons (i.e. expressions 'nil = x', 'x = nil', 'nil != x', and 'x != nil') with 'x' of type C are an alternative way of writing x instanceof C and not(x instanceof C) respectively.

The predicate x instanceof C is translated using the predicate `eval` from the theory of objects. For 'x' of type C, the second argument of this operation is a function, which returns `True` whenever the value of x is constructed using a constructor associated with one of the subclasses of C. For example, the expression 'n != nil', where n is of type Element, amounts to a check that n is in the object store, with the right type. The following HOL code accomplishes this test:

```
eval (os n)
   (λval · case val of AtomicContents refDir showStat URL ⇒ False
                       | ANode name pos ins outs content ⇒ True
                       | CNode name pos ins outs size elements ⇒ True
                       | Link name pos from to ⇒ True)
```

The expression `n` looks up the `object`-typed value associated with oid `n`. The second argument to `eval` is a boolean-valued function); this function returns `True` if the type tag on the value is an Element (i.e., ANode, CNode, or Link). Otherwise, if `n` does not have a binding in `os`, or is bound to an AtomicContents value, then `False` is returned. Any other occurrence of the nil literal is mapped directly to the None-constant, as pointed out in Section 5.5.2.

5.7.2 Attribute Selection

Attribute selections are encoded using the `get` operation. For example, the OASIS expression 'n.incomingLinks', where n is of type Node, is represented as follows:

```
get (os n)
  (λval · case val of AtomicContents refDir showStat URL ⇒ arbitrary
                      | ANode name pos ins outs content ⇒ ins
                      | CNode name pos ins outs size ins ⇒ elements
                      | Link name from pos to ⇒ arbitrary)
```

The second argument to `get` is a function that selects the required incomingLinks attribute if the type tag on the value is `ANode` or `CNode`. For the the wrongly typed cases, i.e. if `n` does not have a binding in `os`, or is bound to a Link or an AtomocContents value, an `arbitrary` value is returned.

5.7.3 Undefined Values

The `arbitrary` constant provides a common way of dealing with undefined function results in HOL. It is, for instance, used to define the head of the empty list in the theory of lists, which is supplied with the standard distribution of Isabelle/HOL. The `arbitrary` constant is of (generic) type α, thus being available for all types. We use the constant for defining the result of accessing attribute values via bad pointer references.

It is important to realize that the rules for managing undefined values in HOL are slightly different than the ones proposed by the ODMG standard. In the ODMG/OQL language proposal, comparison operations (such as $=$, $<$, $>$, $\leq$, $\geq$) with either or both operands being undefined yield *false* as a result, whereas `arbitrary` $=$ `arbitrary` holds *true* in HOL. This may seem surprising, since one could infer, for example, that the 'from' field of two non-existent Links are equivalent. Similar issues already arise with other predefined functions (such as the head of the empty list).

An alternative solution would be to model the result of an expression as an optional value, such that `None` values are used to represent "bad" cases. This representation would have the benefit of allowing us to prove assertions such as "expressions e always terminates normally". However, reasoning about such cases has not been a primary goal of this project, so that the solution based on `arbitrary` values suffices.

5.7.4 Binary Operations on Heterogeneous Collections

OASIS supports heterogeneous collections of objects. Two values can be put in one and the same set as long as a least-upper bound exists of their associated types. HOL is based on the simply typed lambda calculus, which only supports homogeneous collections of values. Nevertheless, it is possible to map OASIS sets to HOL sets in a straightforward manner. OASIS only provides object polymorphism (value polymorphism, particularly for records, is not supported). Since all reference types are represented using the same type of oid values, sets automatically become homogeneous as a result of the mapping.

Binary set expressions in OQL (that is, union, except, and intersect), are also available in Isabelle/HOL. Again, no conflicts occur due to polymorphism. For example, in OQL, one can union a set of objects of type ANode, with a set of type CNode, and the result will be a set of Node objects. In our HOL representation, such a union results in a homogeneous set of oids: both the ANode and CNode set are represented as expressions of type (`oid set`), and the application of the set-union operation ($\cup$) in HOL is well typed.

5.7.5 Bounded Quantifiers

Quantifier expressions are represented using the prefined quantifiers in HOL, which are included in the the standard distribution of Isabelle/HOL. For example, consider the following integrity constraint (we assume two persistent roots nodes and links):

ic : forall c in cnodes: forall e in c.elements: e != nil

The above constraint iterates over the *heterogeneous* collection of all elements of all CNode objects; such an iteration maps to an iteration over a *homogeneous* collection of oid-values in HOL (for the definition of $eltsOf$ and $isElt$, see page 52):

$$\begin{array}{l} ic \equiv \\ \quad \lambda\texttt{os:OS} \cdot \lambda\texttt{cnodes:oid set} \cdot \lambda\texttt{links:oid set} \cdot \lambda\texttt{anodes:oid set} \cdot \\ \qquad \forall \texttt{c} \in \texttt{cnodes} \mid (\forall \texttt{e} \in \texttt{get (os c)}\ eltsOf) \mid \texttt{eval (os e)}\ isElt \end{array}$$

5.7.6 Select-from-where

The select-from-where clause is represented as a predicative set in Isabelle/HOL. The schema translator applies an explicit conversion to sets (using the listtoset conversion operation), if one of the collections in the from-clause turns out to be a list[2] The following OQL query returns the name of all elements of composite nodes which have a position less that 2000:

```
q1 : select e.name
     from c in cnodes, e in c.elements
     where c.position < 2000
```

Predicative sets in HOL may also contain multiple nested iterations. The Isabelle/HOL representation of this query expression is defined as the following predicative set expression (again, the free identifiers in italics font abbreviate the case-splits to encode the required attribute selections):

$$\begin{array}{l} q_1 \equiv \\ \quad \lambda\texttt{os:OS} \cdot \lambda\texttt{cnodes:oid set} \cdot \\ \qquad \{\texttt{get (os}\ e)\ nameOf\} \mid \texttt{c e} \cdot \\ \qquad\quad \texttt{c} \in \texttt{cnodes} \wedge \texttt{e} \in (\texttt{get (os c)}\ eltsOf) \wedge (\texttt{get (os c)}\ posOf) < 2000\} \end{array}$$

5.7.7 Type Casting

OASIS-OQL allows one to down-cast an object o of static type C to a type that is a subtype of C. Down-casting is an unsafe operation, and may result in run-time exceptions. In HOL, the down-casting operation is represented using `get`; the `arbitrary` constant is used to deal with the wrongly typed cases. For example, let us assume the existence of a named root top of type Node. Its value can be an instance of ANode or CNode. Down-casting of top to ANode, using (ANode)top, is represented as follows:

$$\begin{array}{l} q_2 \equiv \\ \quad \lambda\texttt{os:OS} \cdot \lambda\texttt{top:oid} \cdot \\ \qquad \texttt{get (os top)} \\ \qquad\quad (\lambda\texttt{val} \cdot \texttt{case val of AtomicContents refDir showStat URL} \Rightarrow \texttt{arbitrary} \\ \qquad\qquad\qquad\qquad \mid \texttt{ANode name pos ins outs content} \Rightarrow \texttt{arbitrary} \\ \qquad\qquad\qquad\qquad \mid \texttt{CNode name pos ins outs size elements} \Rightarrow \texttt{top} \\ \qquad\qquad\qquad\qquad \mid \texttt{Link name pos from to} \Rightarrow \texttt{arbitrary}) \end{array}$$

[2]This follows the procedure outlined in the ODMG language proposal [CB97], although we map to sets rather than to bags. Recall that bags are not supported in the current Isabelle/HOL release.

The above expression returns the oid value `top`, if top is an instance of ANode, and otherwise the result is undefined (i.e., an `arbitrary` value is returned).

5.8 Mapping of the object manipulation language

In this section, we highlight details of the mapping of commands. Several examples are given to illustrate the use of the generic update operations from Figure 5.1 to encode database specific updates, in particular methods. A complete definition of the translation function $\mathcal{C}$ is found in the Appendix. Below, we explain the general type structure of the output generated by applications of $\mathcal{C}$. The following typing restriction applies, for a given command c of type t and environment $A = [(x_1, t_1), \cdots, (x_n, t_n)]$:

$$\mathcal{C}[\![c]\!]_A :: [\texttt{OS}, \mathcal{T}[\![t_n]\!], \cdots, \mathcal{T}[\![t_1]\!]] \Rightarrow \texttt{OS} \times \mathcal{T}[\![t_1]\!]\ \mathit{option} \times \cdots \times \mathcal{T}[\![t_n]\!]\ \mathit{option} \times \mathcal{T}[\![t]\!]$$

The output is a function, which maps input object store and the program state variables to a tuple.

The tuple in the co-domain type consists of the following components. The first field is a delta (of type `OS`) that encodes the changes that c applies to the population of objects. The n subsequent fields represent the modifications applied to the program state variables (these were ignored in the informal introduction). Changes to variables are registered using options: for modified variables, the new value is returned encapsulated by the `Some`-construct; for variables that are not changed, the constant `None` is returned. The last field of the tuple is the command's return value. In case no return value is used (i.e., the command is of type void), an empty tuple ( ) is returned, which is of type `unit`.

The above representation of commands is slightly simplified, for it ignores object creation. The examples below are of increasing complexity. We start with the representation of basic commands, such as assignments and skip. Object creation is then gradually introduced in the presentation. The idea is that the functional representation of a command c is augmented with a sufficient number of extra parameters, for supplying the oids needed in c.

5.8.1 Skip

The skip operation is represented using the `skip` operation from the theory of objects. For example, consider the following dummy method, defined for Link

```
void dummy1() { skip }
```

In the context of a schema with only one persistent root cnodes (for the examples in this section, we assume that there is only one persistent root), we obtain the following function representation for this method.

```
Link_dummy1 ≡ λos:OS · λcnodes:oid set · λthis:oid ·
   (skip, None, None, ())
```

The name of the function is the same as the name of the method, but the name of the defining class is prepended to avoid name clashes due to overloading. The above function takes three parameters: one for the input object store, one for the cnodes persistent root, and one for the receiver object; since no new objects are created, no extra parameters are needed. The output is a 4-tuple: the first slot in the tuple is the `skip`-delta; the second and third field represent the changes to the variables cnodes and this—no updates to these variables are applied, hence a `None` is returned; the last field represents the return value—skip is of type void, which amounts to an empty tuple '`()`' in the last position.

5.8.2 Assignment

The value of an attribute or variable can be changed using the assignment operations '=', '+=', and '-='. For the representation of these operations in HOL, we distinguish between two cases.

We first consider the case that a new value is assigned to a variable, not to an attribute of an object, and that the expression on the right-hand side is an OQL expression, not a command. For example, the following dummy method is defined for Link:

```
void dummy2(Link p) { p = this }
```

We obtain the following representation of this method in HOL:

$$\begin{aligned}&\mathtt{Link\_dummy2} \equiv \lambda\mathtt{os}{:}\mathtt{OS}\cdot\lambda\mathtt{cnodes}{:}\mathtt{oid\ set}\cdot\lambda\mathtt{this}{:}\mathtt{oid}\cdot\lambda\mathtt{p}{:}\mathtt{oid}\cdot\\&\quad(\mathtt{skip},\mathtt{None},\mathtt{None},\mathtt{Some}(\mathtt{this}),())\end{aligned}$$

The representation is similar to the one for skip, but the position reserved for the variable p now uses the `Some`-contruct to indicate a change.

We now consider the case that a new value is assigned to an attribute of an object, not to a variable. In this case, a side-effect occurs, and we use the `set`-operation from the theory of objects to describe the changes applied to the contents of the store.

In the introduction, we already showed the use of `set` to encode simple attribute updates. Below, we will give an example of an attribute update that shows that the new attribute value can be an arbitrary expression over the input state. Consider the following update method on class Link:

```
void copyIns(Node n) {
    inLinks = n.inLinks
};
```

In Isabelle, we represent this method using the `set`-operation, as follows (the free identifier $inLinksOf$ is the case-split to encode the selection of the inLinks attribute):

```
Node_copyIns ≡ λos:OS · λcnodes:oid set · λthis:oid · λn:oid ·
  (set os this
   (λval ·case val of AtomicContents refDir showStat URL ⇒
                      AtomicContents refDir showStat URL
                     | ANode name pos ins outs content ⇒
                      ANode name pos (get (os n) inLinksOf) outs content
                     | CNode name pos ins outs size elements ⇒
                      CNode name pos (get (os n) inLinksOf) outs size elements
                     | Link name pos from to ⇒
                      Link name pos from to), None, None, None, ())
```

The new value of the attribute clearly depends on the input state. Thus arbitrary OQL expressions can be used on the right-hand side of an assignment operation.

Observe that, if the `this` object is not typed as an ANode or a CNode in the given object store (i.e., it is a Link), the value associated with `this` is not changed. This ensures that object attribute update does not affect the type of an object.

The representation of the other assignment operations ('=', and '+=') is done in a similar fashion, using `set`. It is interesting to point out that attribute assignments expressed using the operations '+=' and '-=' enable the generation of more 'efficient' HOL code. These assignments could also be expressed using the normal assignment operation '=', but this requires an extra look-up of the store to retrieve the old value of the attribute at hand. This is clearly seen from the representation of the above method: the old value of the attribute incomingLinks, which is used to define the new value, does not need to be retrieved explicitly from the store. This would have been the case if the '=' operation is used.

A similar representation in terms of '`set`' is obtained for the cases that the expression on the right-hand side of the assignment is a command, not an expression. The reader is referred to the formal definition in Section 5.8.

5.8.3 Return values

In OASIS-OML, a return statement is used to specify the return value of a method. This statement maps to a tuple in HOL, with the Δ value `skip` in the first position, all subsequent variable positions filled with `None` values, and the return value in the last position. For example, the following method is defined on Link:

```
boolean Link::isConnectedTo(Node n) { return (from == n) or (to == n) }
```

The above method checks whether a given Node object is connected to the receiver Link. We obtain the following HOL representation of this method:

```
Link_isConnectedTo ≡
  λos:OS · λcnodes:oid set · λthis:oid·
     (skip, None, None, (get (os this) fromOf) ∨ (get (os this) toOf))
```

The above function returns a 4-tuple: the return statement does not apply any changes to the store (which amounts to a `skip` value in the first position), or the variables this and cnodes (indicated by the two subsequent None values); the last position specifies the return value. Observe that, the translation algorithm recognizes that from and to are attributes of the receiver, not normal variables.

5.8.4 Object Creation

Object creation is represented using the new construct from the theory of objects. This involves the “invention” of new oids: the oid assigned to a newly created object has to be different from the oids of all other existing objects in the object store. The new oid value is obtained by adding an extra parameter to the command that invokes the creation. For simple object creations, such as in the body of the following initialization method for ANode:

```
ANode create1() { new ANode }
```

only a single new oid needs to be supplied. Assuming no additional variable declarations, the HOL representation of this method is straightforward:

$$\begin{array}{l}\texttt{ANode\_create1} \equiv \\ \quad \lambda\texttt{os}:\texttt{OS}\cdot\lambda\texttt{cnodes}:\texttt{oid set}\cdot\lambda\texttt{this}:\texttt{oid}\cdot\lambda\texttt{nID}:\texttt{oid}\cdot \\ \quad\; (\texttt{new nID}\ (\texttt{ANode}\ \text{“”}\ \texttt{0}\ \{\}\ \{\}\ \texttt{None}), \texttt{None}, \texttt{None}, \texttt{nID})\end{array}$$

Observe that, the attributes of the object are initialized with default values, and that a handle is returned to the newly created object in the last slot of the output tuple. Also observe that, the above representation is quite liberal: any value can be supplied for the parameter nID, even one that is already present in os. The freshness of the extra oid parameters should be enforced at proof-time, by adding suitable preconditions in the goal.

After default initialization of the object attributes, the class constructor is applied if the constructor matches the number of parameters supplied to new. For example, the following method initializes the name and position field of the ANode.

```
ANode create2(string name,int p) {
    new ANode(name,p)
}
```

The HOL representation of this method involves the application of the function ANode_const, which represents the constructor of class Link (the definition of this function is shown in the next paragraph).

$$\begin{array}{l}\texttt{ANode\_create2} \equiv \\ \quad \lambda\texttt{os}:\texttt{OS}\cdot\lambda\texttt{cnodes}:\texttt{oid set}\cdot\lambda\texttt{this}:\texttt{oid}\cdot\lambda\texttt{name}:\texttt{string}\cdot\lambda\texttt{p}:\texttt{int}\cdot\lambda\texttt{nID}:\texttt{oid}\cdot \\ \qquad \texttt{let}\ (\Delta, \texttt{cnodes}_1, \texttt{this}_1, \texttt{name}_1, \texttt{p}_1, \texttt{result}) = \\ \qquad\quad \texttt{ANode\_const}\ (\texttt{smash os}\ (\texttt{new nID}\ (\texttt{ANode}\ \text{“”}\ 0\ \{\}\ \{\}\ \texttt{None})))\ \texttt{cnodes nID name p} \\ \qquad \texttt{in}\ (\Delta, \texttt{cnodes}_1, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{nID})\end{array}$$

The first argument of the function ANode_const takes a smash of the input object store and the delta that represents the newly created object (with default values for its attributes). The ‘receiver’ object of the constructor is the newly created object; this is reflected by the third argument of the function (i.e., the value nID). The constructor may update the persistent variable cnode (this is seen from the second field in the output tuple). Changes to other variables are discarded after the constructor is applied (this is seen from the sequence of None values in the output tuple).

5.8.5 Sequential Composition

The sequential composition operation is represented using the smash-operation on deltas. For example, consider the following constructor for ANode:

ANode(string n, int p) { name = n; position = p };

The code shown below illustrates how the sequential composition in the constructor-body is represented in terms of the representation of 'name = n' and 'position = n'.

```
Link_const ≡ λos:OS · λcnodes:oid set this:oid · λn:string · λp:int·
  let $(\Delta_1, \texttt{cnodes}_1, \texttt{this}_1, \texttt{n}_1, \texttt{p}_1, \texttt{rtn}_1) = \mathcal{C}[\![\text{name} = \text{n}]\!]_A$ os cnodes this n p;
      $(\Delta_2, \texttt{cnodes}_2, \texttt{this}_2, \texttt{n}_2, \texttt{p}_2, \texttt{rtn}_2) = \mathcal{C}[\![\text{position} = \text{n}]\!]_A$ (smash os $\Delta_1$)
          (case $\texttt{cnodes}_1$ of None ⇒ cnodes | Some y ⇒ y)
          (case $\texttt{this}_1$ of None ⇒ this | Some y ⇒ y)
          (case $\texttt{n}_1$ of None ⇒ n | Some y ⇒ y)
          (case $\texttt{p}_1$ of None ⇒ p | Some y ⇒ y)
  in (smash $\Delta_1$ $\Delta_2$,
      case $\texttt{cnodes}_2$ of None ⇒ $\texttt{cnodes}_1$ | Some y ⇒ y,
      case $\texttt{this}_2$ of None ⇒ $\texttt{this}_1$ | Some y ⇒ y,
      case $\texttt{n}_2$ of None ⇒ $\texttt{n}_1$ | Some y ⇒ y,
      case $\texttt{p}_2$ of None ⇒ $\texttt{p}_1$ | Some y ⇒ y, $\texttt{rtn}_2$)
```

In the above definition, the variable Δ_1 represents the modifications of the object store applied by the first step of the constructor. The changes applied in the second step (Δ_2) are obtained by applying the representation function of 'name = n' to the the intermediate state 'smash os Δ_1' instead of os, thus taking the modifications of the store generated by the first step into account. The effects of both steps are accumulated using a smash of Δ_1 and Δ_2. The variable A is the environment, needed as an argument to $\mathcal{C}$; here we have:

$$A = [(\text{p}, \text{int}), (\text{n}, \text{string}), (\text{this}, \text{ANode}), (\text{cnodes}, \text{set}\langle\text{CNode}\rangle)]$$

Changes applied to variables in the program state are accumulated using case-splits. This is done in a straightforward manner: if a variable is changed in the first step, the modified value is used as input for the second step in the sequence; if a variable is modified in the second step, that modified value is returned, otherwise the value resulting from the first step is returned.

The return value of a sequential composition of commands $c_1; c_2$ is the value returned by c_2. This is seen in the example code above: the function returns a tuple with the return value $\texttt{rtn}_2$ of the second step in the last position.

Creation of objects in one or both steps of the sequence is dealt with as follows. Let n and m be the number of new oid parameters of the representation function of commands c_1 and c_2, respectively. The representation function of the command sequence '$c_1; c_2$' takes a number of $n + m$ new oid parameters, of which the first n are passed to the representation function of c_1, and the others are passed to the representation function of c_2.

5.8.6 Local variable declaration

Variable declarations are mapped to let-constructs in HOL. For example, consider the following command sequence

int m(int x){int z = 10; return x+z}

The representation of the above command sequence is defined as follows (we assume an initial environment A with only two variables, namely the method receiver and the parameter):

$$\begin{array}{l} m \equiv \lambda \mathtt{os{:}OS} \cdot \lambda \mathtt{this{:}oid} \cdot \lambda \mathtt{x{:}int} \cdot \\ \quad \mathtt{let}\ (\Delta, \mathtt{thisa}, \mathtt{xa}, \mathtt{za}, \mathtt{rtn}) = \mathcal{C}[\![\text{return x + z}]\!]_{(\mathtt{z},\mathtt{int})::A}\ \mathtt{os}\ \mathtt{this}\ 10 \\ \quad \mathtt{in}\ (\Delta, \mathtt{thisa}, \mathtt{xa}, \mathtt{rtn}) \end{array}$$

In the above definition, the let-construct supplies the initial value for `z`. The (possibly) updated value of the variable is dropped from the program state in the '`in`' clause.

5.8.7 Conditional branch

Conditional updates are represented using the predefined Isabelle/HOL construct for conditionals "`if-then-else`". Consider the following conditional method defined on Node:

```
vooid removeIncomingLink(Link k) {
   if (k in incomingLinks) then { incomingLinks -= set(k) }
   else skip
};
```

For the HOL representation of this method, we obtain the following definition (the representation of the attribute assignment is abbreviated using a function h, of which the definition is not shown):

$$\begin{array}{l} \mathtt{Node\_removeIncomingLink} \equiv \lambda \mathtt{os{:}OS} \cdot \lambda \mathtt{cnodes{:}oid\ set} \cdot \lambda \mathtt{this{:}oid} \cdot \lambda \mathtt{k{:}oid} \cdot \\ \quad \mathtt{if}\ \mathcal{E}[\![\text{k in incomingLinks}]\!]_A \\ \quad \mathtt{then}\ \mathcal{C}[\![\text{incomingLinks -= set(k)}]\!]_A\ \mathtt{os}\ \mathtt{cnodes}\ \mathtt{this}\ \mathtt{k} \\ \quad \mathtt{else}\ \mathcal{C}[\![\text{skip}]\!]_A\ \mathtt{os}\ \mathtt{cnodes}\ \mathtt{this}\ \mathtt{k} \end{array}$$

It is important to note that the if, then and else branch all take the same incoming database state. Recall that the occurrences of `os` and `k` in the translation of k in incomingLinks are captured by the λ-bindings in the definition of the function `Node_removeIncomingLink` (see Section 5.7).

The representation function of the conditional clause takes a number of $n + m$ new oid parameters, if a number of n and m new oid parameters is required for the then and else clause, respectively. The first n parameters are passed to the representation of the then-clause; the last m parameters are passed to the else clause.

5.8.8 Method call

Parameter passing to methods is done as in Java, using "call by value". It is conveniently encoded in terms of the function representation of methods. For example, consider the following method, which invokes the method create2 from the previous examples.

```
ANode create3(string n,int p) { this.create2(n,p) }
```

The application of the method maps to a function application in HOL. We obtain the following representation for the above method definition.

$$
\begin{array}{l}
\texttt{ANode\_create3} \equiv \\
\quad \lambda\texttt{os:OS} \cdot \lambda\texttt{cnodes:oid set} \cdot \lambda\texttt{this:oid} \cdot \lambda\texttt{n:string} \cdot \lambda\texttt{p:int} \cdot \lambda\texttt{nID:oid} \cdot \\
\quad\;\; \texttt{let } (\Delta, \texttt{cnodes}_1, \texttt{this}_1, \texttt{n}_1, \texttt{p}_1, \texttt{result}) = \texttt{ANode\_create2 os cnodes this n p} \\
\quad\;\; \texttt{in } (\Delta, \texttt{cnodes}_1, \texttt{None}, \texttt{None}, \texttt{None}, \texttt{result})
\end{array}
$$

The above definition shows how call-by-value semantics works: any updates applied to the method parameters are discarded and a `None` value is returned in the output tuple, whereas updates applied to a persistent root, or attributes of an object are preserved.

Late binding of update methods is dealt with in the same way as is done for retrieval methods, using `if-then-else` and `eval`. The reader is refered to Section 5.4, for an example.

5.8.9 Bounded Iteration

The bounded iteration construct introduces a form of parallelism in our update language. The construct is represented using the generic function `apply`. The following transaction removes a value from the contents of all objects in the anodes collection. For this example, we assume the presence of one root anodes.

```
set⟨ANode⟩ removeAtomicContent(AtomicContents ac) {
  foreach a in anodes where (ac in a.content) do
       a.content -= set(ac)
}
```

The object is removed from the content field of all ANode objects in the anodes persistent root. The following function represents this transaction:

```
removeAtomicContent $\equiv$
  $\lambda$os:OS $\cdot$ $\lambda$anodes:oid set $\cdot$ $\lambda$ac:oid $\cdot$
    let $A = \{\mathtt{a} \mid \mathtt{a} \in \mathtt{anodes} \wedge \mathtt{ac} \in (\mathtt{get}\ (\mathtt{os\ a})\ \mathit{contentOf})\}$;
        $\Delta_f = (\lambda \mathtt{a} \cdot (\lambda \mathtt{os\ anodes\ ac\ a} \cdot (\mathtt{set\ os\ a}\ \mathit{contentMod}, \mathtt{None}, \mathtt{None}, \mathtt{None}, ()))$
                    $\mathtt{os\ anodes\ ac\ a})$
    in $(\mathtt{apply}\ A\ (\lambda \mathtt{a} \cdot \mathtt{fst}(\Delta_f\ \mathtt{a})), \mathtt{None}, \mathtt{None}, ())$
```

Informally, applications of this function return a 4-tuple. The first field of this tuple is a Δ value that holds the modified ANode objects. The Δ is expressed using the `apply`-operation. The first argument of `apply` represents the (restricted) set that is the subject of the iteration. Observe that this set is bound before the iteration begins (i.e., we use the input state `os` for the interpretation of this set). The second argument to `apply` is a function that applies the iteration variable '`a`' to each element of the set. The definition of `apply` deals with conflicting parallel updates, as discussed in Section 5.5.

The second and third field of the output tuple represent the changes to the root anodes and the variable ac. These variables are declared outside the scope of the loop, and cannot be modified. Hence, for their positions in the output tuple, a `None` value is returned.

The last field of the output tuple represents the return value of the loop (in this case, no value is returned).

An additional difficulty with set iterations arises from object creation. The invocation of the new construct within a loop requires the allocation of a statically unknown number of new oids. For example, consider the following transaction for initialization of the database.

```
set⟨Node⟩ initDB() {
  foreach d in set(struct(name:“X125”,pos:500),
                   struct(name:“X126”,pos:510),
                   struct(name:“X127”,pos:520)) do
      d.create2(d.name,d.pos)
}
```

The transaction creates three new ANode objects. The method application in each step of the iteration returns a handle to the newly created object; these handle values are collected and returned as a set by the iteration construct (unlike the previous example, where no value was returned).

For each step of the set iteration, a new oid needs to be invented. In general, however, it is not known statically how many new oids are needed: the number of objects that is created depends on the size of the collection that is iterated over. In HOL, we deal with such cases by using an abstract oid generator function. Instead of parameterizing the delta representation functions with an exact number of new oids, now a function is used, which for each element of the set, generates a new oid value. For the above example transaction, we obtain the following HOL representation.

```
initDB ≡
 λos:OS · λcnodes:oid set · λnID:string × int ⇒ oid ·
   let A = {(“xxx”, 500), (“yyy”, 510), (“zzz”, 520)};
       Δ_f = (λd · (λos cnodes d nID ·
        let (Δ, cnodes, n_1, p_1, res) = ANode_create2 os cnodes fst(d) snd(d) nID
        in (Δ, cnodes, None, res)) os cnodes d (nID x));
       ret = {(let (Δ, cnodes_1, d_1, ret) = Δ_f d in ret) | a · a ∈ A}
   in (apply A (λd · fst(Δ_f) d), None, ret)
```

Here, the ‘nID’ parameter is a function of type string × int ⇒ oid. This function should provide a fresh oid for each element in the set of record values (recall from the previous section that, records are represented using unlabeled tuples in HOL). Such a function can be postulated as a precondition in proofs. The automatic generation of such preconditions (e.g. for transaction safety) is discussed at length, in the later chapters of this thesis.

Nesting of loops may lead to curried functions for the new oid parameters. The introduction of a loop, simply adds an additional variable dependency to all new oid parameters of the command body of the loop.

Another difference with the previous example is that the accumulation of the values that result from each step in the iteration. The returned values are accumulated in a set, as is seen in the extra ret clause in the let construct.

5.9 Model-inherent constraints

In the previous sections, it was shown how OASIS constraints and methods are represented as functions in HOL. In this section, we discuss some additional implicit assertions that arise from the schema and its representation in HOL. These so-called model-inherent constraints need to be asserted explicitly, as preconditions in proofs to be performed about the schema.

OASIS class specifications are statically type checked before they are translated to HOL. Static type checking ensures that when one object references another, that reference is either nil or it occurs in the object store with the right type. These implicit assertions about *referential integrity* should be specified explicitly as preconditions in proofs. Figure 5.9 gives an example of how we specify such constraints for a given object store ‘os’, in the context of the case study schema. For instance, it asserts that for each Link object in the store, the ‘from’ attribute value should either be nil (represented as a None-value) or it should occur in ‘os’ with the right type (i.e. Node).

Object creation is one of the complicating factors in reasoning about object-oriented software specifications. It involves the implicit generation of new oids, which must be made explicit in the formal representation. As pointed out in the previous sections, oid generation is taken care of by the translation functions: the function representation of a method is augmented with a sufficient number of extra parameters, to supply the new oid values. These extra parameters have to be instantiated with appropriate values, if proofs are to be performed about the method. Actually, we are not interested in assigning specific values for new oids. The instantiation can be done in a more *abstract* manner. For example, the function representation of a method creating just one new object is augmented with one extra parameter of

```
∀x∀y | os x = Some y →
 get (os x) (λval · case val of AtomicContents refDir showStmt URL ⇒ True
                                  | ANode name pos ins outs content ⇒
                                     (∀u ∈ ins | u ≠ None → eval (os u) isLink)∧
                                     (∀u ∈ outs | u ≠ None → eval (os u) isLink)
                                  | CNode name pos ins outs size elements ⇒
                                     (∀u ∈ ins | u ≠ None → eval (os u) isLink)∧
                                     (∀u ∈ outs | u ≠ None → eval (os u) isLink)∧
                                     (∀u ∈ elements | u ≠ None → eval (os u) isElt)
                                  | Link name pos from to ⇒
                                     (from ≠ None → eval (os from) isNode)∧
                                     (to ≠ None → eval (os to) isNode)
```

Figure 5.9: Example: referential integrity assertion

type oid. In the goal, this parameter will be filled in (by our goal generator) with the value ‘Some nID’, where ‘nID’ is a (free) variable, that stands for an unspecified oid value. Sufficient preconditions about the freshness of these values with respect to a given object store are specified declaratively in proof goals. Two properties need to be asserted:

- the new oid should not already be used in the input state os, nor
- should it be equal to any other new oid.

These requirements are expressed quite easily. Continuing with our example, we simply add the condition $(\texttt{Some nID}) \notin (\texttt{oids os})$ as a precondition to the goal. The second requirement is only relevant if the method creates a number of new objects (say n). In this case, it should be asserted that $(\texttt{Some nID}_i) \neq (\texttt{Some nID}_j)$ for each $i, j \leq n$.

Notice that, the above treatment of new oids in proofs is highly declarative in nature: the “freshness” of new oids only needs to be specified declaratively by including sufficient preconditions in the goal. Such a declarative description works fine since the reasoning of the theorem prover relies on symbolic manipulation (nothing needs to be actually computed).

5.10 Related work

Object-oriented specification methodologies and object-oriented programming have become increasingly important in the past ten years. Not surprisingly, this has recently led to an interest in object-oriented program verification in the theorem prover community, mainly using higher-order logic (HOL). Several different approaches to modeling object-oriented features in HOL have been presented [San97, JvdBH$^+$98, NW98]. We give a brief survey and comparison of the different approaches. We also compare our work to the work of Sheard and Stemple [SS89], who present a system for reasoning about relational database systems.

For the representation of updates as *deltas*, we compare our work to the work of Doherty and Hull [DH95].

Sheard and Stemple [SS89] present an approach to automatically verify that a database transaction preserves a number of static integrity constraints. Relational database schemas are translated to an extension of Boyer and Moore-style logic with higher-order functions. A basic theory of numbers, lists, records, and finite sets is implemented from scratch in Lisp. Update statements are modeled as generic higher-order operations in the logic, which take functions to make them specific. So called *meta-lemmas* are derived about these operations for the task of transaction safety analysis, using *term-rewriting*[3]. This approach also underlies the embedding of OASIS in HOL: OASIS operations for manipulation and retrieval of data from the persistent object store are modeled as generic higher-order functions in HOL, and an equational theory of rewrite rules is derived about the interaction of these operations. Our work extends the work of Sheard and Stemple. It deals with an object-oriented specification framework and develops a theory of shared objects in HOL, in addition to the standard data type theories. The embedding of OASIS in HOL is essentially an extension of the basic HOL data type theories with a theory of objects. The comparison with the work of Sheard and Stemple reveals ten years of progress in the field of automated theorem proving: Sheard and Stemple built a proof system from scratch, starting with higher-order functions and axiomatization of the basic data types. Using an off-the-shelf higher-order logic theorem prover, such as PVS or Isabelle, this is no longer needed today.

Santen [San97] uses Isabelle/HOL to reason about class specifications in Object-Z. The work considers abstract *algebraic specifications* of methods, thus considering an assertional approach, in contrast to the imperative method definitions considered in our work.

For the representation of class schemas, [San97] use a *value semantics*, in contrast to the *reference semantics* that is needed in our work. Reference semantics deals with imperative object-oriented features, such as object identities, aliasing, object sharing, and recursive class definitions (the distinction between reference and value semantics is discussed in Section 2.3).

Naraschewski and Wenzel [NW98] present an encoding of object-oriented classes , based on a shallow embedding of extensible records with structural subtyping in Isabelle/HOL. Similar to [San97], the approach is also algebraic. Using extensible records, method specifications are translated to polymorphic functions in HOL. The receiver object is modeled as an extensible record, with a ρ- variable (see [CM94]) that can match any unknown number of extra fields (to allow instantiation with an instance of a subclass). Their theory of records could be used to replace our (unlabeled) tuple representation of structs.

Jacobs *et al* [JvdBH$^+$98] define a translation of Java classes to PVS higher-order logic, based on co-algebras. This work is closest related to our work. Their framework also deals with a reference semantics, but there are some important differences because our work considers a database language, not a general purpose programming language. First, there is no rich expression language ("query language") in Java that deals with operations on sets, lists, etc. Sets and lists are modeled as objects in Java, which have reference semantics, in contrast to the value semantics that is used for these types in our database language (see Section 2.3.2). In our work, sets and lists are built-in complex values which map to similar

[3]Isabelle uses natural deduction, in addition to term-rewriting

structures in HOL. Second, their work considers non-termination, run-time exceptions, and it supports general forms of recursion and while loops. In database programming languages, these features are often ignored, for reasons of efficiency (see [Qia93]). The OASIS update language cannot express all queries and updates. Its bounded-iteration construct with parallel execution semantics is similar to the construct discussed in [Qia93], which differs from loops in Java.

Our model of objects builds on the delta value ideas of Doherty and Hull [DH95]. A delta represents a difference between database states. We use deltas to formalize the effects of primitive update operations on the persistent object store. In [DH95], delta values are used to describe proposed updates in the context of cooperative work; deltas are first-class citizens, available for manipulation in the user language. In our work, delta values are used to cope with set-oriented updates; they are found in the semantics representation, not in the user language.

5.11 Summary

We discussed a translation of OASIS object-oriented database schemas to higher-order logic (HOL). The translation to HOL provides a foundation for semantics-based reasoning, using the Isabelle theorem prover. A generic theory of objects is defined, which serves to express the semantics of different concrete database schemas.

The generic theory of objects is an orthogonal extension of the basic HOL data types theories, to enable reasoning about object-oriented systems. This theory provides a model of the store. The generic operations defined in this theory are used to encode specific method definitions. The encoding handles non-trivial features of object-oriented programming, such as late binding, heterogeneous collections of objects and inheritance. The *soundness* of the translation to HOL is not examined formally, since this requires the definition of a meta theoretic framework which covers both the OASIS and HOL language. Instead, the correctness of the definitions will be validated experimentally, by an implementation of the semantics functions in ML. This turns out to be far more productive, since the implementation is also used to examine case studies for transaction safety, compensation, and commutativity analysis.

Chapter 6

Implementation of the OASIS/HOL embedding in ML

In this chapter, we discuss an ML[Pau96] implementation of the OASIS/HOL embedding. The implementation is used to validate the correctness of the formal translation discussed in the previous chapter. Validation of the translation algorithms is accomplished by running the implementation on several (large) example schemas; the rigorous type checking of the output that is performed by Isabelle, validates the *soundness* of the translation. More importantly, the ML implementation supports the experimental work discussed in the later chapters of this thesis. Automatic code generation is a prerequisite for quick experimentation with different database schemas. The generation of HOL code from a simple example may already yield highly complex code, and the manual translation of larger examples is not feasible. An example is included in Section 6.6, to illustrate the complexity of the output that is generated.

6.1 Architecture of the tool

Figure 6.1 gives an overview of the architecture of the tool. A *schema translator* is used to translate the database schema to the higher-order logic of Isabelle. This component implements the formal translation discussed in Chapter 5. The input is an OASIS schema, in ascii form; initial parsing using Lex and Yacc converts the input to an internal abstract syntax tree (AST) representation in ML. The output is an Isabelle theory file that gives definitions for the class structures, methods, transactions, and constraints defined in the schema. The resulting theory file is then loaded into an Isabelle session, together with the generic theory of objects (which provides a database-independent model of the object store, see Section 5.5). The user can now start to prove desired properties about the schema, e.g., to prove that a given

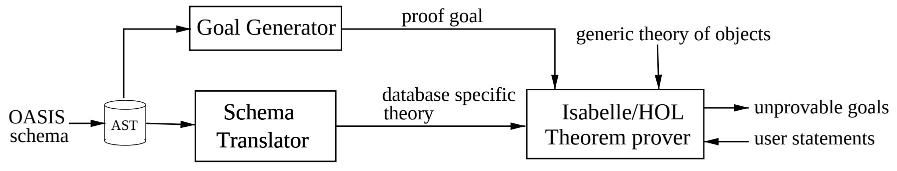

Figure 6.1: Architecture of the tool

constraint is an invariant of a certain method.

The *goal generator* eliminates the need for the user to enter complex proof goals by hand. It defines high-level ML functions that automate the generation of the required proof goals. For example, there is a function that generates the proof goal that is needed to verify that a given constraint is an invariant of a certain method. The user can then start trying to prove this goal, by applying tactics.

The implementation of the schema translator and the goal generator is done in ML. There are two reasons why ML is chosen for the implementation. First of all, the implementation of inductive definitions is done quite naturally in a functional language; the mapping of semantics equations to actual code is almost done in a "one-to-one" manner, using pattern matching in ML. This assures that the ML implementation is close to the formal definition. Second, ML is the meta language of the Isabelle system. Programming the compiler in ML is beneficial, e.g. for developing pretty printing functions that require information about the schema.

The rest of this chapter is organized as follows. Section 6.2 briefly introduces programming in ML. Section 6.3 discusses the translation from concrete to abstract syntax. The data types for the representation of the abstract syntax tree of a schema in ML are given. Section 6.4 discusses a mini-type-checker. Section 6.5 discusses the Isabelle code generator. An example of a translation of a method is given in Section 6.6. Finally, conclusions are given in Section 6.7.

6.2 Higher-order programming in ML

This section introduces the main concepts of the ML (functional) programming language. It is only a very short summary. Interested readers may find more details in [Pau96].

ML is a *functional programming* language; *pattern matching* is one of its characteristic features. Recursive functions over data types are simply defined by just enumerating the possible patterns. For example, consider the definition of the sum of a list of integer values. One can distinguish two cases; i.e., the sum of the empty list which is 0, and if the list is non-empty, the sum is the head plus the sum of the tail. In ML, this definition is expressed declaratively as follows:

```
fun sum [] = 0
  | sum (a::A) = a+(sum A);
> val sum = fn : int list -> int
```

The last line is printed by the ML interpreter, and gives the (inferred) type of the function. If we apply this function to the list `[10,5,6]`, we get the (obvious) result:

```
sum [10,5,6];
> 21 : int
```

Empty versus non-empty is the most common sort of case analysis for lists. ML supports *exceptions*, to define partial functions. A typical example of exception programming in ML is the function hd, to extract the head of a list. We define this function by declaring an exception *Hd*, as follows.

```
exception Hd;
fun hd [] = raise Hd
  | hd (a::A) = a
> val hd = fn : 'a list -> 'a
```

If we apply this function to the empty list, an exception of type `Hd` is raised. Exceptions are captured by associating exception handlers with functions.

Using ML's *polymorphic types* and *higher-order functions*, it is possible to define highly generic operations. As an example, consider an operation `merge` on lists:

```
fun merge f [] b = b
  | merge f (a::A) b = f(a,(merge f A b));
> val merge = fn : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b
```

The definition of `merge` is done by pattern matching on the list in the second argument: the first pattern defines the merge operation when applied to an empty list; the second pattern is for non-empty lists. The generic type of this function is inferred by ML: it includes two type variables α (written as `'a`) and β (written as `'b`), which can be instantiated with arbitrary types.

Many operations can be defined in terms of this generic `merge` operation. For example, consider the following function definitions:

```
fun sum A = merge (fn(a,b)=>a+b) A 0;
fun implode A = merge (fn(a,b)=>a^b) A "";
```

The definition of '`sum`' provides an alternative for the previous definition of '`sum`'. The function '`implode`' for concatenating a list of strings is defined using the *same* operation `merge`. Note that, the type variables α and β are instantiated differently, for the definition of `sum` and `implode`.

ML is mainly for functional programming, but it also allows limited forms of imperative programming, using so-called *reference variables*. These are similar to pointers in imperative programming languages. Reference variables should be used with care. The reader is referred to [Pau96] for more details. The implementation of the OASIS/HOL embedding uses a reference variable to represent the abstract syntax tree of the "current" database schema.

6.3 Translation from concrete to abstract syntax

The OASIS to HOL compiler front-end is written in C++ using Lex and Yacc. It transforms the concrete syntax of an OASIS schema definition to an abstract syntax tree representation in ML. In this section, we define the ML data structure that is used to represent the abstract syntax tree. Only a brief summary is given.

The required definitions for abstract syntax are stored in the ML module '`syntax.ML`'. The abstract syntax tree type is the list type '`definition list`', where '`definition`' is a data type that enumerates the different possible entries of an OASIS schema:

```
datatype definition
   = Class of string*string*string*(operation)list
   | Name of string*typ
   | Constraint of (string*exp)
   | Transaction of (typ*string*(string*typ)list*com) list
```

For example, the constructor `Class` is used to represent class definitions. The constructor takes a 4-tuple of class *type* (a string which indicates whether the class is abstract or concrete), the class name (also a string), the name of the parent class, and a list of *operations*. Operations are attribute declarations, constructor definitions, retrieval method definitions, update method definitions, and abstract method declarations. The type `operation` is declared as follows:

```
datatype operation
   = attribute of typ*string
   | umethod of typ*string*(string*typ)list*com
   | rmethod of typ*string*(string*typ)list*exp
   | amethod of typ*string*(string*typ)list
   | constructor of string*(string*typ)list*com;
```

The other data types, for types, expressions, and commands are defined in a similar fashion. These definitions yield a disambiguated OASIS grammar. For example, consider a fragment of the definition of the data type of expressions (not all expressions are listed)

```
datatype exp = TRUE
             | FALSE
             | INT of int
             | VAR of string
             | STRING of string
             | CHAR of char
             | EQ of exp*exp
             | NEQ of exp*exp
             | STRUCT of (string*exp)list
             | SET of (exp list)
             | ...
```

Observe that, in the abstract syntax all operators are represented using prefix notation. For example, the ML expression '`NEQ(VAR "x",INT 10)`' is of type '`exp`' and represents the OASIS boolean expression '`x != 10`'. The `SET` construct takes a (possibly empty) list of sub-expressions: these correspond to the elements of an enumerated set.

6.4 A mini-type-checker in ML

Information about the minimum type of expressions and commands is needed for the translation to HOL. The ML module '`type-checker.ML`' implements a mini-type-checker for OASIS, which type-checks a schema based on the abstract syntax tree representation. The type-checker is a boolean-valued function on the abstract syntax tree, which type-checks all methods, transactions and integrity constraints in the current schema.

The module defines several auxiliary functions for type checking, e.g. to infer the unique type of an expression, or command; to compute the least-upper bound of two types; and to determine whether one type is a subtype of another. Inductive definitions for these functions are given using pattern matching in ML. The current schema, which is needed for the implementation of most of these functions is made implicit, using a *reference variable*.

As an example, consider a small fragment of the definition of the function `tc_exp`, for type checking OQL expressions.

```
fun tc_exp(TRUE,env)  = Bool
  | tc_exp(FALSE,env) = Bool
  | tc_exp(EQ(e1,e2),env) =
        (let val t1 = tc_exp(e1,env) ;
             val t2 = tc_exp(e2,env) ;
             val t = LUB(t1,t2)
         in Bool end)
...
> val tc_exp = fn : exp * (string*typ)list -> bool
```

The function `tc_exp` implements the typing rules for expressions (as shown in Appendix C. The definition is inductive, by pattern matching on the first argument. The second argument represents the environment (as a list of pairs), which maintains information about the free variables in the expression. Note the third rule for equality, which involves an invocation of the function `LUB`, which is another ML function of type `typ*typ -> typ` for the computation of a least upper bound of the types supplied as an argument. A function `TypeCheck` is defined at the top level. It checks all methods, transactions, constructors and integrity constraints in the current schema.

6.5 Isabelle/HOL back-end

The type-checker module can be extended with different back-ends to perform actual code generation. An Isabelle/HOL back-end has been implemented as part of this research project, but other code generators might be implemented as well.

The Isabelle/HOL back-end defines a function to generate an Isabelle theory and ML file from the abstract syntax tree. The output theory and ML file are a string, which are directed to a text file (with extension `.thy` and `.ML` respectively).

The Isabelle/HOL code is generated by an ML implementation of the semantics functions $\mathcal{E}$ (for OQL expressions), $\mathcal{C}$ (for commands), and $\mathcal{T}$ (for types), as defined in Appendix D. The ML definitions have the same inductive structure as the formal definitions. For example, the following fragment of code shows part of the implementation of $\mathcal{E}$:

```
fun tr_exp(TRUE,env) = "True"
  | tr_exp(EQ(e1,e2),env) = "("^tr_exp(e1,env)^" = "^^tr_exp(e2,env)^")"
  | tr_exp(FORALL(x,A,p),env) =
      "(! "^x^":"^tr_exp(A,env)^". "
      ^tr_exp(A,case tc_exp(p,env)
                  of Set(t) => (x,t)::env
                   | _ => raise tc_error("collection type expected"))^")"
...
> val tr_exp = fn : exp * (string*typ)list -> string
```

The definition is done by pattern matching on the first argument (which is the OQL expression that is translated). The second argument is the environment, with information about the variables. Patterns are shown for the expressions true, $e_1 = e_2$, and forall x in A : p

The second pattern defines the translation of $e_1 = e_2$, directly in terms of the translation of e_1 and e_2 (the function ^ performs string concatenation). The translation of the universal quantification, as shown in the third pattern, is slightly more difficult. For the translation of the predicate p, we add the variable x paired with its type t to the environment. The type of x is obtained by application of the function `tc_exp` as shown above.

6.6 Use of the schema translator

The schema translator is typically used as follows. Consider the schema definition of the SEPIA example shown in Appendix A. We assume that the schema is stored in the text file `SEPIA.oa`. The schema is translated in three steps. First, the abstract syntax tree representation is generated by typing the following at the Unix command line:

```
oasis SEPIA.oa > SEPIA.oa.ML
```

The above command invokes the front-end compiler and redirects the the output abstract syntax tree representation of the schema to the ML text file `SEPIA.oa.ML`. Now, from the ML command line, we can load this tree into an Isabelle session by typing the following command:

```
> use "SEPIA.oa.ML";
```

The file `SEPIA.oa.ML` contains a statement for initialization of the reference variable for the abstract syntax tree representation of the current schema. It subsequently invokes the function `TypeCheck` from the type-checker module to type check the operations in the schema. If all operations are type correct, the `.thy` and `.ML` file are generated. The output is as follows (not all methods are listed):

```
[opening SEPIA.oa.ML]
val it = () : unit
Checking constructor AtomicContents
Checking method Node::addIncomingLink
Checking method Node::isConnectedTo
....
...
Checking constructor Link
Checking method Link::isConnectedTo
Checking constraint c1
Checking constraint c2
Checking constraint c3
Checking constraint c4
Checking constraint c5
val it = true : bool
Generating SEPIA.thy file...
Generating SEPIA.ML file...
val it = () : unit
```

Finally, the resulting Isabelle theory is loaded into an Isabelle session, using the command:

```
> use_thy "SEPIA";
```

After Isabelle has type-checked the schema, the user can start trying to prove desired properties of the schema.

The process of type-checking the schema and the generation of the HOL files requires only a few seconds, although the file with extension `.thy` contains highly complex Isabelle/HOL code. To give the reader an impression of the complexity of the output, we only give the translation of one of the methods removeElement which is defined on CNode:

```
boolean removeElement(Element n) {
   if (n != nil) and (n in elements) and
     (forall x in elements : not(x.isConnectedTo(n))) then {
       elements -= set(n); return true
   } else return false
};
```

The method invokes the method isConnectedTo which is declared as abstract in Element, and has different implementations for Node and Link. The automatically generated Isabelle/HOL code for this method is huge:

```
CNode_removeElement_def "CNode_removeElement == %(os::OS) (cnodes::(oid set))
 (links::(oid set)) (anodes::(oid set)) (this::oid) (n::oid). (if (((eval (os
 n) (%val. case val of AtomicContents referenceDirectory showStatement URL =>
 False |ANode name position incomingLinks outgoingLinks content => True | CNo
de name position incomingLinks outgoingLinks size elements => True | Link nam
e position from to => True))) & ((n : (get (os this) (%val. case val of Atomi
cContents referenceDirectory showStatement URL => arbitrary | ANode name posi
tion incomingLinks outgoingLinks content => arbitrary | CNode name position i
ncomingLinks outgoingLinks size elements => elements | Link name position fro
m to => arbitrary))))) & ((! x:(get (os this) (%val. case val of AtomicConten
ts referenceDirectory showStatement URL => arbitrary | ANode name position in
comingLinks outgoingLinks content => arbitrary | CNode name position incoming
Links outgoingLinks size elements => elements | Link name position from to =>
 arbitrary)). ˜( (if (eval (os x)(%val. case val of AtomicContents referenceD
irectory showStatement URL => False | ANode name position incomingLinks outgo
ingLinks content => False | CNode name position incomingLinks outgoingLinks s
ize elements => False | Link name position from to => True)) then (let (delta
,cnodes,links,anodes,n,this1,result) = Link_isConnectedTo os cnodes links ano
des (x) n in result)  else (let (delta,cnodes,links,anodes,n,this1,result) =
Node_isConnectedTo os cnodes links anodes (x) n in result))))) then (%(os::OS
) (cnodes::(oid set)) (links::(oid set)) (anodes::(oid set)) (this::oid) (n::
oid). let (delta,cnodes1,links1,anodes1,this1,n1,result) = (%(os::OS) (cnodes
::(oid set)) (links::(oid set)) (anodes::(oid set)) (this::oid) (n::oid). let
 (delta,cnodes1,links1,anodes1,this1,n1,result) = (%(os::OS) (cnodes::(oid se
t)) (links::(oid set)) (anodes::(oid set)) (this::oid) (n::oid). let (delta,c
nodes1,links1,anodes1,this1,n1,result) = (%(os::OS) (cnodes::(oid set)) (link
s::(oid set)) (anodes::(oid set)) (this::oid) (n::oid). (skip,None::(oid set)
 option,None::(oid set) option,None::(oid set) option,None::oid option,None::
oid option,())) os cnodes links anodes this n in (delta,cnodes1,links1,anodes
1,this1,n1,(%(os::OS) (cnodes::(oid set)) (links::(oid set)) (anodes::(oid se
t)) (this::oid) (n::oid). insert n {}) (smash os delta)  (case cnodes1 of Non
e=>cnodes | Some y=>y) (case links1 of None=>links | Some y=>y) (case anodes1
 of None=>anodes | Some y=>y) (case this1 of None=>this | Some y=>y) (case n1
 of None=>n | Some y=>y))) os cnodes links anodes this n in (smash delta (set
 os (this) (%val. case val of AtomicContents referenceDirectory showStatement
 URL => AtomicContents referenceDirectory showStatement URL | ANode name posi
tion incomingLinks outgoingLinks content => ANode name position incomingLinks
 outgoingLinks content | CNode name position incomingLinks outgoingLinks size
 elements => CNode name position incomingLinks outgoingLinks size ((elements)
 - result) | Link name position from to => Link name position from to)),None:
:(oid set) option,None::(oid set) option,None::(oid set) option,None::oid opt
ion,None::oid option,())) os cnodes links anodes this n in (delta,cnodes1,lin
ks1,anodes1,this1,n1,(%(os::OS) (cnodes::(oid set)) (links::(oid set)) (anode
s::(oid set)) (this::oid) (n::oid). True) (smash os delta)  (case cnodes1 of
None=>cnodes | Some y=>y) (case links1 of None=>links | Some y=>y) (case anod
es1 of None=>anodes | Some y=>y) (case this1 of None=>this | Some y=>y) (case
 n1 of None=>n | Some y=>y))) os cnodes links anodes this n else (%(os::OS) (
cnodes::(oid set)) (links::(oid set)) (anodes::(oid set)) (this::oid) (n::oid
). let (delta,cnodes1,links1,anodes1,this1,n1,result) = (%(os::OS) (cnodes::(
oid set)) (links::(oid set)) (anodes::(oid set)) (this::oid) (n::oid). (skip,
None::(oid set) option,None::(oid set) option,None::(oid set) option,None::oi
d option,None::oid option,())) os cnodes links anodes this n in (delta,cnodes
1,links1,anodes1,this1,n1,(%(os::OS) (cnodes::(oid set)) (links::(oid set)) (
anodes::(oid set)) (this::oid) (n::oid). False) (smash os delta)  (case cnode
s1 of None=>cnodes | Some y=>y) (case links1 of None=>links | Some y=>y) (cas
e anodes1 of None=>anodes | Some y=>y) (case this1 of None=>this | Some y=>y)
 (case n1 of None=>n | Some y=>y))) os cnodes links anodes this n)"
```

We will not attempt to explain this code. The above example should only convince the reader that code generation is virtually impossible to do by hand. Also, it gives an indication of the complexity that has to be dealt with in the later chapters, if proofs are to be performed about such methods.

6.7 Summary

In this chapter, we discussed an implementation of the OASIS/HOL embedding in ML. One of the goals of the implementation was to validate the correctness of the formal definitions, in particular soundness. Indeed, the ML implementation revealed several mistakes in the formal definition of the semantics functions. These ranged from minor typos, to more fundamental errors of generating wrongly typed HOL code. However, mistakes such as these could be identified by the Isabelle type checker , by testing the output for various examples schemas.

The implementation was roughly done within a period of 3 weeks, and consists of 2049 lines of ML code. The ML implementation is meant as a research prototype, not as an end-user product. Several improvements can be made. As yet, the type-checker only performs a rough check of an OASIS schema, and e.g. rules for method overriding are not implemented yet.

The implementation of the OASIS/HOL compiler provides a helpful tool for the practical verification work discussed in the rest of this thesis. The implementation enables us to quickly experiment with several different OASIS database schemas, as well as revisions to the same schema.

Chapter 7

Verification support for invariant analysis

An analysis technique to mechanically verify that the methods defined in a database schema do not violate the specified integrity constraints (invariants) is presented. The technique builds on the formal representation of the object-oriented database schema in higher-order logic, discussed in Chapter 5. It is shown how the Isabelle theorem prover, and its machinery for automated theorem proving in higher-order logic, can be used for automated invariant analysis. Practical experiments demonstrate the feasibility of the approach.

7.1 Approaches to the preservation of invariants

Efficiently maintaining the integrity of large collections of persistent data is one of the challenges of database technology. In the context of databases, various different methods have been proposed to deal with the efficient maintenance of integrity constraints (e.g., see [FP93] for an overview). The following sections give highlights of the basic characteristics distinguishing the three approaches: integrity checking, integrity verification, and integrity enforcement. Important concepts are introduced that are used later.

7.1.1 Integrity checking

One way to enforce database integrity is by testing at *run-time* those constraints that are possibly violated by a transaction before allowing the transaction to commit. This is usually called *integrity checking*. A naive approach would be to check the constraints *after* each update is applied, and perform a rollback operation if a constraint violation is detected. Often, this method is not feasible in practice because the testing of a constraint and the potential rollback operation are difficult to implement efficiently [Qia88, BGL96].

Several different methods have been proposed to limit the amount of run-time transaction overhead due to integrity constraint checking (e.g., [Nic82, HI85, Law95]). The common goal of these methods is to derive a "minimal" test that can be evaluated *before* the transaction is executed, thus avoiding expensive rollback operations. The test should at least be a *sufficient precondition* for the transaction; i.e., if the test succeeds on the input state, it should follow that the state after the update is consistent. Most methods employ, however, Dijkstra's notion of *weakest precondition* [Dij76], which provides a necessary and sufficient test for correct transaction execution. This avoids the unnecessary rejection of updates (as may be the case when a sufficient condition is used). Typically, an initial weakest precondition is derived from the syntax of the transaction and constraint. This condition is then further simplified under the assumption that the constraint holds in the state before the transaction. As pointed out in [BGL96], this "is a fundamental idea underlying many algorithms for the automatic maintenance of integrity constraints". Indeed, the idea of simplifying the test, by assuming that the database state is consistent before the update, is exploited by most of the existing techniques for integrity checking (e.g., [Qia88, Law95, BGL96, Sel95]). Having made this assumption, it is possible to significantly improve the efficiency of the test. For instance, consider the insertion of a tuple John, and the constraint that all persons should earn at least 5000 pesos. If we assume that the constraint is satisfied before the update, it only needs to be checked that John earns more that 5000 pesos afterwards, rather than checking all persons.

Traditionally, the notion of consistency is tied to the notion of database transaction. That is, transactions are the atomic units of operations that should preserve the constraints. In the context of object-oriented databases, it is far less clear how integrity of the data should be managed. One of the reasons is that the traditional notion of global integrity constraint somehow contradicts with the dominant *object-centered* position that is taken; as pointed out by Jagadish and Qian ([JQ92]),

> "*Control in object-oriented databases is localized rather than centralized—there is no centralized place where constraints can be stated, reasoned about, and maintained. Instead, every object is responsible to maintain the constraints attached to it with respect to changes to its attributes.*"

Most commercial object-oriented databases merely ignore the issue of integrity by not offering any facilities for conceptual constraint specification. Also, the Object Database Management Group (ODMG) does not include facilities for general declarative constraint specification in its standard: the data model only addresses key and referential integrity constraints (see [CB97]).

If constraints *are* supported, an important question is when they should be checked. Jagadish and Qian ([JQ92]) make a distinction between so-called *hard* and *soft* constraints. Hard constraints are checked after each method call, whereas the checking of soft constraints is deferred until the transaction commits. Soft constraints are similar to an ANSI/ISO SQL92 standard ([SQL92]) feature called "*deferrable integrity constraints*", as opposed to "*immediate constraints*" which are checked after each individual SQL statement.

7.1.2 Integrity verification

In order to avoid expensive rollback operations and run-time validation of constraints, one could also attempt to verify at compile-time that a transaction will *never* lead to an inconsistent database state, provided that the initial state was valid. In other words, one can also attempt to *prove* that the specified transactions are *safe* with respect to the constraints. This is called *integrity verification*. For instance, consider the modification of a tuple John, by updating its salary, and a constraint that all persons should be born after 1900. Clearly, the constraint is never violated by the update.

Several different formal and semi-formal methods have been proposed for integrity verification, e.g. using Hoare Logic [GM79], Boyer-Moore Logic [SS89], and abstract interpretation [BS96, BS97]. Tool support in this area may rely on some form of automated theorem proving (e.g., [SS89, BS97]).

It is important to note that, in case of integrity verification, it is the programmer's responsibility to include sufficient tests in the code (typically by inserting appropriate if-then-else statements), rather than that the system attempts to infer these checks. This technique is known as *defensive programming*. Whenever safety of an update cannot be proved, modifications should be made in the code until correctness is achieved. At this point it is helpful if the system could suggest to the designer how the transaction might be corrected, either by adding tests or extra updates. In [SMS87], guidelines are given for extracting such feedback from the unproved subgoals left by the theorem prover. Integrity verification *subsumes* integrity checking, whenever this kind of feedback is provided.

7.1.3 Integrity enforcement

An important question to be answered by a constraint maintenance facility is how to fix the problem when a constraint violation is detected. Traditionally, the response of a database management system to an integrity violation is to rollback the transaction. An alternative approach is to modify the original transaction such that integrity is preserved. In this case, additional repairing actions should bring the database back to a consistent state. This is called *integrity enforcement* (for an overview, see [FP93]). The modification of the transaction may be performed at compile-time, or at run-time. Active databases provide an effective framework for implementing *run-time* integrity enforcement by means of so-called ECA (event-condition-action) rules: the repairing action (action-part) is executed only if a constraint violation (event) is detected. An integrity verification method can assist *compile-time* integrity enforcement, by providing good feedback facilities to illuminate the missing repairing actions.

7.1.4 Our approach

In this chapter, our focus is on static integrity verification. We will concentrate on verifying methods with respect to globally specified integrity constraints. We assume that the constraints are *hard* (see Section 7.1.1): they should be valid after each method call. This thesis concentrates on the verification of hard integirty constraints and methods, rather than on transactions and soft constraints. We consider *inter-object constraints* [JQ92], which apply across objects, and multi-step methods that can affect more than one object. We do not consider transactions, which typically involve sequences of method calls. Effectively, this provides the same complexity as the transaction and soft constraint combination.

7.2 Invariant analysis using the Isabelle theorem prover

Verifying that a method preserves an invariant is a difficult task. As pointed out in [BGL96], it is undecidable to check if a given transaction preserves a given integrity constraint so that "*one cannot hope to have an algorithm for checking this, even for simple transactions and simple constraint specification languages*". A theorem prover such as Isabelle can be used, however, to *partly* automate this task. Isabelle is an interactive theorem prover, but it has a suite of powerful tactics for automatically solving goals in higher-order logic. A general purpose verification strategy is available for HOL, which attempts automated reasoning, leaving any unproved subgoals to the user. This strategy is the basis of the invariant analysis technique presented here.

Isabelle's default proof strategy combines term-rewriting and natural deduction, using its *Simplifier* and *Classical Reasoner* tools (see Chapter 4). These tools support the usual data types, e.g. int, bool, string, list, and set. Since those data types directly correspond to similar OASIS data types, we get a substantial amount of support for automated reasoning for free. Extensions are made to support the theory of objects (see Section 5.5), which is used to encode methods and constraints. In short, the invariant analysis technique is the standard Isabelle/HOL verifier with extra theorems installed for supporting the object representations presented in Chapter 5.

We can distinguish the following steps in the verification process:

1. Formulation of the proof goal;
2. Normalization of the goal;
3. Depth-first proof search.

These steps are carried out successively, without human interaction. Any goals that cannot be solved automatically may be examined further using Isabelle's interactive proof mode, or used for feedback extraction. Below, we discuss details of each of these steps.

7.2.1 Step 1: Formulation of the goal

Verifying that a method implementation preserves an invariant amounts to proving a formal theorem in HOL. In the verification, we assume that the constraint holds in the input database state and attempt to prove that the constraint still holds after the application of the method, for arbitrary input database state and method parameter values. Such a requirement on the implementation of a method can de specified as a *proof goal* in Isabelle. The generation of this goal, for a given method and constraint, is automated by two ML functions: `method_safety_goal` and `start_proof`, which are implemented by the *goal generator* (see Figure 6.1). The use of these functions and the structure of the resulting goal is discussed below.

To verify that the removeElement method, defined in class CNode preserves the constraint ic defined in Section 5.7.5, we type the following at the Isabelle command prompt:

```
- start_proof(method_safety_goal("removeElement","CNode","ic",[]));
```

On typing the above command, Isabelle responds with the proof goal shown below.

```
Level 0
(eval (os this) isCNode)∧
(ic os cnodes links anodes)
→  let (Δ, cnodes₁, links₁, anodes₁, this₁, n₁, result) =
          CNode_removeElement os cnodes links anodes this n
    in ic (smash os Δ)
          (case cnodes₁ of None ⇒ cnodes | Some y ⇒ y)
          (case links₁ of None ⇒ links | Some y ⇒ y)
          (case anodes₁ of None ⇒ anodes | Some y ⇒ y)
```

The goal is in the form of an implication (→), where the constraints are assumed to hold in the input database state ‘os’ (as seen in the premise), and we attempt to prove that the constraint still holds after the application of the method (as stated in the conclusion), for arbitrary database state os, persistent root values (cnodes, anodes, and links), receiver (this), and parameter value (n).

The definitions of the method and constraint are generated by a schema translator, as discussed in Chapter 6. The method is represented by the CNode_removeElement-function. The definition of this function (and more explanation) is found on page 51; here we only discuss its signature. For a given input object store (os), persistent root values (cnodes, anodes, and links), receiver object (this) and parameter value (n), the function returns a 7-tuple: this includes the modifications to the object store in the first field, the modifications to the persistent root variables, and the modifications to method parameters (including the implicit parameter this). The last field is the return value of the method.

The modifications to the object store are encoded as a Δ-value. The use of Δ-values is a characteristic feature of our theory of objects (see Section 5.5). The Δ-value returned by the function CNode_removeElement in the goal is a small object store that only has bindings for the objects that are modified by the method. To “commit” these local changes to the input object store, we use the smash-operator (see Section 5.5): the expression ‘(smash os Δ)’ in the above goal represents the object store after the application of the method.

The modifications to the persistent roots are *by value-result*: case-expressions are used to pass on possible changes to the persistent roots. For each persistent root, the old value is passed if no changes are applied (indicated by the None-tag); the modified value is passed if changes have been made (indicated by the Some-tag). Details of the encoding of changes to variables are found in Section 5.8.

The goal includes several additional assumptions. A typing condition is added for the this-reference. Other assumptions (not shown in the example above) deal with referential integrity and the assignment of new oids. The reader is referred to Section 5.9 for details.

7.2.2 Step 2 : Normalization of the goal

The analysis starts by applying simplifications to the goal, using term-rewriting. Simplifications are applied to the term ‘C(smash os Δ)’ in the conclusion of the goal. The main purpose of these simplifications is to to embed the update (Δ) into the constraint predicate C such that a much simpler test is obtained that can be evaluated before the update is applied. This test is called the *updated constraint* [HI85]; such a test effectively provides a weakest precondition for the update. The updated constraint may be further simplified under the assumption that the constraint holds prior to the update (as discussed in Section 7.1.1, this is a fundamental idea underlying many algorithms for integrity checking, e.g., see [HI85, Qia88, BGL96, LT93, Law95]).

The derivation of the updated constraint and its simplification under the assumption that the constraint holds before the update are implemented using the Isabelle Simplifier (see Section 4.2.4). The Simplifier performs term-rewriting with a set of theorems of the following form:

$$[|H_1\,;\,\cdots\,;\,H_n|] \Rightarrow LHS = RHS$$

Such theorems are read as conditional rewrite rules: a term unifying with the expression on the left-hand side of the equality sign (LHS) is rewritten to the term that appears on the right-hand side (RHS), provided that the hypotheses $(H_1, \ldots, H_n)$ hold. The default Isabelle Simplifier installs a large collection of standard reduction rules for HOL; new rules can be added to customize the Simplifier to particular tasks. At present, 49 new rewrite rules have been added to support the theory of objects. The most important ones are listed in Section 5.5.4. In this section we demonstrate the practical use of these rules in combination with standard rewrite rules, for the default data type theories.

The analysis starts by expanding the definitions of the method and the constraint. The HOL representation of the constraint and the Δ-value of the method are:

$$\begin{array}{rcl} \Delta & \equiv & \texttt{if}\ \mathit{cond}\ \texttt{then}\ (\texttt{set os this}\ \mathit{eltMod})\ \texttt{else skip} \\ C(\texttt{os}) & \equiv & \forall \texttt{c} \in \texttt{cnodes} \mid \forall \texttt{e} \in \texttt{get}\ (\texttt{os c})\ \mathit{eltsOf} \mid \texttt{eval}\ (\texttt{os e})\ \mathit{isElt} \end{array}$$

The representation of the constraint is discussed in (see the definition of ic on page 62). A detailed explanation of the definition of the method, including the Δ-value and the definitions of the abbreviations in italics font, is found on page 51. The opening of these definitions in the goal, expands the conclusion of the goal '$C(\texttt{smash os}\ \Delta)$' to the following formula:

$$\begin{array}{l} \forall \texttt{c} \in \texttt{cnodes} \mid \\ \quad \forall \texttt{e} \in \texttt{get}\ ((\texttt{smash os}\ (\ \texttt{if}\ \mathit{cond}\ \texttt{then set os this}\ \mathit{delElt}\ \texttt{else skip}))\ \texttt{c})\ \mathit{eltsOf} \mid \\ \qquad\quad \texttt{eval}\ ((\texttt{smash os}\ (\texttt{if}\ \mathit{cond}\ \texttt{then set os this}\ \mathit{delElt}\ \texttt{else skip}))\ \texttt{e})\ \mathit{isElt} \end{array}$$

The above formulae evaluates the constraint in the state that results after the application of the removeElement method. Various rewritings can be performed to simplify this formula. The goal typically contains patterns of the form:

$$\texttt{get}\ ((\texttt{smash os}\ \Delta))\ x)\ f \qquad \text{and} \qquad \texttt{eval}\ ((\texttt{smash os}\ \Delta))\ x)\ p$$

which are rewritten using the rules listed in Figures 5.5 and 5.4 to yield the updated constraint.

The Simplifier first splits one of the conditionals. Splitting of conditionals is a built-in feature of the Simplifier, which makes use of the rule:

$$P(\texttt{if}\ Q\ \texttt{then}\ x\ \texttt{else}\ y) \leftrightarrow ((Q \rightarrow P(x)) \wedge (\neg Q \rightarrow P(y)))$$

By splitting the conditional, the Simplifier rewrites the example to:

$$\begin{array}{l} \mathit{cond} \rightarrow \\ \quad (\forall \mathtt{c} \in \mathtt{cnodes} \mid \forall \mathtt{e} \in \mathtt{get}\ ((\mathtt{smash\ os}\ (\mathtt{set\ os\ this}\ \mathit{delElt}))\ \mathtt{c})\ \mathit{eltsOf} \mid \\ \qquad \mathtt{eval}\ ((\mathtt{smash\ os}\ (\mathtt{if}\ \mathit{cond}\ \mathtt{then\ set\ os\ this}\ \mathit{delElt}\ \mathtt{else\ skip}))\ \mathtt{e})\ \mathit{isElt}) \wedge \\ \neg \mathit{cond} \rightarrow \\ \quad (\forall \mathtt{c} \in \mathtt{cnodes} \mid \forall \mathtt{e} \in \mathtt{get}\ ((\mathtt{smash\ os}\ (\mathtt{skip}))\ \mathtt{c})\ \mathit{eltsOf} \mid \\ \qquad \mathtt{eval}\ ((\mathtt{smash\ os}\ (\mathtt{if}\ \mathit{cond}\ \mathtt{then\ set\ os\ this}\ \mathit{delElt}\ \mathtt{else\ skip}))\ \mathtt{e})\ \mathit{isElt}) \end{array}$$

This formula is then further simplified using an advanced feature of the Simplifier called “local assumptions”. That is, in simplifying the conclusion of an implication, the tool uses additional rewrite rules it extracts from the assumption. This is done using the *congruence rule* for implication:

$$[\!|\ P_1 \leftrightarrow Q_1\ ;\ Q_1 \Rightarrow P_2 \leftrightarrow Q_2\ |\!] \Rightarrow (P_1 \rightarrow P_2) \leftrightarrow (Q_1 \rightarrow Q_2)$$

Given this rule, the Simplifier assumes Q_1 (i.e., the rewritten form of P_1) and extracts rewrite rules from it when simplifying P_2. The `else`-branch of the if-then-else in the conclusion of the implication on the left-hand side of the conjunction is eliminated using this rule: the Simplifier rewrites the conclusion using the rewrite ‘$\mathit{cond} \equiv$ `True`’ rule it extracts from the assumption. The right-hand side of the conjunction in the formula is rewritten in a similar fashion; in this case, the `then`-branch is eliminated using the local assumption ‘$\mathit{cond} =$ `False`’. We obtain the simplified formula:

$$\begin{array}{l} \mathit{cond} \rightarrow (\forall \mathtt{c} \in \mathtt{cnodes} \mid \forall \mathtt{e} \in \mathtt{get}\ ((\mathtt{smash\ os}\ (\mathtt{set\ os\ this}\ \mathit{delElt}))\ \mathtt{c})\ \mathit{eltsOf} \mid \\ \qquad \mathtt{eval}\ ((\mathtt{smash\ os}\ (\mathtt{set\ os\ this}\ \mathit{delElt}))\ \mathtt{e})\ \mathit{isElt}) \wedge \\ \neg \mathit{cond} \rightarrow (\forall \mathtt{c} \in \mathtt{cnodes} \mid \forall \mathtt{e} \in \mathtt{get}\ ((\mathtt{smash\ os}\ (\mathtt{skip}))\ \mathtt{c})\ \mathit{eltsOf} \mid \\ \qquad \mathtt{eval}\ ((\mathtt{smash\ os\ skip})\ \mathtt{e})\ \mathit{isElt}) \end{array}$$

The Simplifier performs further simplification of this formula. It first rewrites the expression ‘`smash os skip`’ to ‘`os`’ (using the rule labeled **[smash_right_id]** in Figure 5.2). The result of this simplification is that the quantifier expression in the conclusion has become identical to $C(\mathtt{os})$. Since $C(\mathtt{os})$ was assumed to hold in the goal, the Simplifier rewrites this term to `True` (again, using the congruence rule for implication shown abobe). The resulting formula is then further simplified using the standard rewrite rules for the boolean data type $P \rightarrow \mathtt{True} = \mathtt{True}$ and $P \wedge \mathtt{True} = P$. We arrive at the following simplified formula:

$$\begin{array}{l} \mathit{cond} \rightarrow \\ \quad (\forall \mathtt{c} \in \mathtt{cnodes} \mid \forall \mathtt{e} \in \mathtt{get}\ ((\mathtt{smash\ os}\ (\mathtt{set\ os\ this}\ \mathit{delElt}))\ \mathtt{c})\ \mathit{eltsOf} \mid \\ \qquad \mathtt{eval}\ ((\mathtt{smash\ os}\ (\mathtt{set\ os\ this}\ \mathit{delElt}))\ \mathtt{e})\ \mathit{isElt}) \end{array}$$

The conclusion of this formula is further simplified (in a few steps) as shown below. In step (1), the Simplifier applies the rules labeled **[get_set]** (see Figure 5.5) and **[eval_set]** (see Figure 5.4), which results in two if-then-else patterns. In step (2), the function composition $\mathit{isElt} \circ \mathit{delElt}$ is rewritten to isElt; this reduction follows by simple case-function composition (see the definitions of these functions on page 51 and page 52). Further rewriting is performed using the standard rule ‘`if` P `then` e `else` $e = e$’. Finally, in step (3) the conditional is split, after which the `else` case is removed under the assumption that $C(\mathtt{os})$ holds.

$$(\forall \texttt{c} \in \texttt{cnodes} \mid \forall \texttt{e} \in \texttt{get}\ ((\texttt{smash os}\ (\texttt{set os this}\ \mathit{delElt}))\ \texttt{c})\ \mathit{eltsOf} \mid \\ \quad \texttt{eval}\ ((\texttt{smash os}\ (\texttt{set os this}\ \mathit{delElt}))\ \texttt{e})\ \mathit{isElt})$$

$$\overset{(1)}{\equiv} (\forall \texttt{c} \in \texttt{cnodes} \mid \forall \texttt{e} \in (\ \texttt{if this} = \texttt{c} \land \texttt{this} \in \texttt{oids}(\texttt{os}) \\ \quad \texttt{then get}\ (\texttt{os c})\ (\mathit{eltsOf} \circ \mathit{delElt}) \\ \quad \texttt{else get}\ (\texttt{os c})\ (\mathit{eltsOf}))\ \mid \\ \quad \texttt{if this} = \texttt{c} \land \texttt{this} \in \texttt{oids}(\texttt{os})\ \texttt{then eval}\ (\texttt{os c})\ (\mathit{isElt} \circ \mathit{delElt}) \\ \quad \texttt{else eval}\ (\texttt{os c})\ (\mathit{isElt}))$$

$$\overset{(2)}{\equiv} (\forall \texttt{c} \in \texttt{cnodes} \mid \forall \texttt{e} \in (\ \texttt{if this} = \texttt{c} \land \texttt{this} \in \texttt{oids}(\texttt{os}) \\ \quad \texttt{then get}\ (\texttt{os c})\ (\mathit{eltsOf} \circ \mathit{delElt}) \\ \quad \texttt{else get}\ (\texttt{os c})\ (\mathit{eltsOf}))\ \mid \texttt{eval}\ (\texttt{os c})\ (\mathit{isElt}))$$

$$\overset{(3)}{\equiv} (\forall \texttt{c} \in \texttt{cnodes} \mid (\texttt{this} = \texttt{c} \land \texttt{this} \in \texttt{oids}(\texttt{os})) \rightarrow \\ \quad \forall \texttt{e} \in (\texttt{get}\ (\texttt{os c})\ (\mathit{eltsOf} \circ \mathit{delElt})) \mid \texttt{eval}\ (\texttt{os e})\ \mathit{isElt})$$

No further rewriting can be performed, and we are left with the following simplified goal:

$$\boxed{\begin{array}{l} \texttt{Level}\ 1 \\ 1.\ (\texttt{eval}\ (\texttt{os this})\ \mathit{isCNode}) \land \\ \quad (\forall \texttt{c} \in \texttt{cnodes} \mid \forall \texttt{e} \in (\texttt{get}\ (\texttt{os c})\ \mathit{eltsOf}) \mid \texttt{eval}\ (\texttt{os e})\ \mathit{isElt}) \land \mathit{cond} \rightarrow \\ \quad\ (\forall \texttt{c} \in \texttt{cnodes} \mid (\texttt{this} = \texttt{c} \land \texttt{this} \in \texttt{oids}(\texttt{os})) \rightarrow \\ \quad\quad \forall \texttt{e} \in (\texttt{get}\ (\texttt{os c})\ (\mathit{eltsOf} \circ \mathit{delElt})) \mid \texttt{eval}\ (\texttt{os e})\ \mathit{isElt}) \end{array}}$$

The above example demonstrates how much we can benefit from standard machinery of the theorem prover in the analysis of invariants. As seen in the example above, complex rewritings are performed by the Simplifier to simplify the test '$C(\texttt{smash os}\ \Delta)$' under the assumption that '$C(\texttt{os})$' holds.

7.2.3 Step 3 : Depth-first proof search

Term rewriting is often not sufficient to prove that a method preserves an invariant. The simplified goal is analyzed further using Isabelle's Classical Reasoner (see Chapter 4). The *Classical Reasoner* implements several "naive" proof search algorithms (such as a depth-first and a best-first search) based on a given set of introduction and elimination rules. The default configuration of the tool installs such rules for sets, lists, booleans, etc. Extensions are provided for the theory of objects. This amounts to adding the introduction and elimination rules shown in Figure 5.6 and Figure 5.7.

We use the depth-first search algorithm of the Classical Reasoner. This handles substantial proofs: variables introduced by the use of quantifiers are automatically instantiated, and backtracking is performed to handle alternative unifiers. Between deduction steps, the Simplifier is invoked to attempt further rewriting. Using this proof search algorithm, the proof of the example goal is found within seconds, although many detailed steps are performed behind the scenes. Some of the important steps in the proof are discussed below.

The Classical Reasoner first applies all possible *safe* inference steps. Safe inference steps are the ones that do not introduce any new unknown terms, or instantiate variables in the goal (hence, no backtracking is needed). In case of the example, the tool resolves the simplified goal with the introduction rule for implication (the rule labeled $(\rightarrow I)$, see page 42), followed by a two-times application of the introduction rule for set-bounded universal quantification (the rule labeled $(\forall_{\in} I)$, see page 42). Furthermore, the variable ‘`this`’ is replaced by ‘`c`’, using the assumption ‘`this` = `c`’ as a substitution rule; we obtain:

```
Level 2
1.!! c e.
   [| eval (os c) isCNode; c ∈ oids(os); cond
      ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
      c ∈ cnodes; e ∈ get (os c) (eltsOf ∘ delElt) |] ⇒ eval (os e) isElt
```

Not shown is the expansion of $cond$, which does not play a role in the rest of the proof. The set-bounded universal quantifiers in the assumption are not eliminated, because their elimination is *unsafe* (see Section 4.2.3 for details). Before any unsafe steps are tried, the Classical Reasoner first applies the safe elimination rules **[evalE]** (see Figure 5.7) and **[oidsE]** (see Figure 5.6), which are added by the theory of objects. The elimination of ‘`eval` (`os c`) $isCNode$’ results in a list of four subgoals, one for each dynamic type of ‘`c`’ :

```
Level 3
1.!! c e refDir showStmt URL.
   [| os c = Some(AtomicContents refDir showStmt URL); False; os c ≠ None;
      ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
      c ∈ cnodes; e ∈ get (os c) (eltsOf ∘ delElt) |] ⇒ eval (os e) isElt
2.!! c e name pos ins outs content.
   [| os c = Some(ANode name pos ins outs content); False; os c ≠ None;
      ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
      c ∈ cnodes; e ∈ get (os c) (eltsOf ∘ delElt) |] ⇒ eval (os e) isElt
3.!! c e name pos ins outs size elements.
   [| os c = Some(CNode name pos ins outs size elements); True; os c ≠ None;
      ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
      c ∈ cnodes; e ∈ get (os c) (eltsOf ∘ delElt) |] ⇒ eval (os e) isElt
4.!! c e name pos from to.
   [| os c = Some(CNode name pos from to); False; os c ≠ None;
      ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
      c ∈ cnodes; e ∈ get (os c) (eltsOf ∘ delElt) |] ⇒ eval (os e) isElt
```

Although all the (four) different type cases are present, during proofs the irrelevant subgoals (identified by `False`-valued cases—see Section 5.7) are immediately dropped from the proof state. For the example, only one subgoal is relevant (namely the case for `CNode`):

```
Level 4
1.!! c e name pos ins outs size elements.
   [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
      ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
      c ∈ cnodes; e ∈ get (os c) (eltsOf ∘ delElt) |] ⇒ eval (os e) isElt
```

At this stage, no further safe inference steps can be done. Before, any unsafe steps are tried, the Classical Reasoner first invokes the Simplifier to attempt further rewriting. For the example, the expression ‘`get (os c)` $(\mathit{eltsOf} \circ \mathit{delElt})$’ in the assumptions of the goal is rewritten as follows:

$$\begin{aligned} & \texttt{get (os c)}\ (\mathit{eltsOf} \circ \mathit{delElt}) \\ \overset{(1)}{\equiv}\ & \texttt{get (Some(CNode name pos ins outs size elts))}\ (\mathit{eltsOf} \circ \mathit{delElt}) \\ \overset{(2)}{\equiv}\ & (\mathit{eltsOf} \circ \mathit{delElt})\ \texttt{(CNode name pos ins outs size elts)} \\ \overset{(3)}{\equiv}\ & \texttt{elts} - \{n\} \end{aligned}$$

In step (1), the Simplifier uses the assumption ‘`os c` $=$ `Some(CNode(`$\cdots$`)`’ as a rewrite rule. In step (2), the rule labeled **[get_Some]** in Figure 5.5 is applied; the resulting expression ‘`elts - {n}`’ follows after β-conversion in step (3). The following simplified goal is obtained:

```
Level 5
1.!! c e name pos ins outs size elts.
    [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
       ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
       c ∈ cnodes; e ∈ elts − {n} |] ⇒ eval (os e) isElt
```

The process of applying safe inference steps and rewriting is repeated until a fixedpoint is reached. In this case, one more safe inference step is applied, namely the elimination of the set difference (see page 46); we obtain:

```
Level 6
1.!! c e name pos ins outs size elts.
    [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
       ∀c ∈ cnodes | ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
       c ∈ cnodes; e ∈ elts; e ∉ {n} |] ⇒ eval (os e) isElt
```

No more safe steps or rewritings apply to this goal. There is no other option than to apply the unsafe rule $(\forall_{\in} E)$ (see page 42). The elimination of the quantifier results in two subgoals:

```
Level 7
1.!! c e name pos ins outs size elts.
    [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
       (?t c e name pos ins outs size elts) ∉ cnodes;
       c ∈ cnodes; e ∈ elts; e ∉ {n} |] ⇒ eval (os e) isElt
2.!! c e name pos ins outs size elts.
    [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
       ∀e ∈ (get (os (?t c e name pos ins outs size elts)) eltsOf) |
             eval (os e) isElt;
       c ∈ cnodes; e ∈ elts; e ∉ {n} |] ⇒ eval (os e) isElt
```

The variable $?t$ is a function unknown, which may depend on the variables that are quantified in the goal. Isabelle solves the first goal by contradiction. This leads to an instantiation of ‘$?t$ `c` $\cdots$ `elts`’ to ‘c’:

```
Level 8
1.!! c e name pos ins outs size elts.
   [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
      ∀e ∈ (get (os c) eltsOf) | eval (os e) isElt;
      c ∈ cnodes; e ∈ elts; e ∉ {n} |] ⇒ eval (os e) isElt
```

Using the assumption 'os c = Some(CNode(···)' as a rewrite rule, the Simplifier now rewrites the expression 'get (os c) *eltsOf*' to 'elts', in a similar way as was shown above. The subsequent elimination of the second quantifier expression results in a list of two subgoals:

```
Level 9
1.!! c e name pos ins outs size elts.
   [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
      (?t c e name pos ins outs size elts) ∉ elts;
      c ∈ cnodes; e ∈ elts; e ∉ {n} |] ⇒ eval (os e) isElt
2.!! c e name pos ins outs size elts.
   [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
      eval (os (?t c e name pos ins outs size elts)) isElt;
      c ∈ cnodes; e ∈ elts; e ∉ {n} |] ⇒ eval (os e) isElt
```

Again, the first goal solves by contradiction, which instantiates the variable (in this case, '?t c ··· elts' is instantiated to e):

```
Level 10
1.!! c e name pos ins outs size elts.
   [| os c = Some(CNode name pos ins outs size elements); os c ≠ None;
      eval (os e) isElt;
      c ∈ cnodes; e ∈ elts; e ∉ {n} |] ⇒ eval (os e) isElt
```

The remaining goal solves by assumption: the conclusion of the goal unifies with one of the assumptions. Isabelle responds with the following message to indicate that the proof has been found:

```
Level 11
No subgoals!
```

7.3 Experimental results

The automated proof strategy is used to verify that the methods in the example schema do not violate the specified integrity constraints. The example schema includes 6 class definitions, 13 method definitions, and 6 constraints. It has to be verified that each method in the schema preserves the constraints. Thus 78 theorems need to be verified. Table 7.1 shows timings for the automated verification of these theorems using the automated analysis technique discussed in the previous sections. All proof times are in seconds, with Isabelle running on a SUN 296 MHz Ultra-SPARC-II, under Solaris. The times given are only a rough guide of the efficiency of the automated verification. As expected, the times indicate that the trivial proofs

METHOD / CONSTRAINT	c_1	c_2	c_3	c_4	c_5	c_6
`Element::changeNameTo`	2.3s.	1.8s.	1.8s.	2.8s.	2.8s.	2.0s.
`Node::addIncomingLink`	1.6s.	1.0s.	1.0s.	2.3s.	1.4s.	3.5s.
`Node::addOutgoingLink`	1.5s.	1.0s.	1.0s.	2.3s.	1.5s.	1.5s.
`Node::removeIncomingLink`	1.3s.	0.7s.	0.7s.	2.5s.	1.2s.	2.6s.
`Node::removeOutgoingLink`	1.3s.	0.7s.	0.7s.	2.3s.	1.2s.	1.1s.
`Node::isConnectedTo`	0.6s.	0.4s.	0.4s.	1.1s.	0.7s.	0.7s.
`ANode::createAtomicContentsIn`	7.5s.	5.3s.	5.1s.	14.7s.	11.5s.	7.7s.
`ANode::removeAtomicContent`	1.6s.	1.1s.	1.1s.	2.6s.	1.6s.	1.6s.
`CNode::createANodeIn`	9.1s.	4.3s.	5.0s.	13.3s.	59.1s.	5.5s.
`CNode::createLinkIn`	15.3s.	10.5s.	9.4s.	78.0s.	118.3s.	12.8s.
`CNode::removeElement`	3.4s.	1.0s.	1.0s.	161.8s.	109.7s.	2.6s.
`CNode::createCNodeIn`	14.2s.	6.2s.	6.2s.	46.7s.	154.5s.	9.0s.
`Link::isConnectedTo`	0.7s.	0.4s.	0.4s.	1.1s.	0.8s.	0.7s.

Table 7.1: Verification times(in seconds) for invariant analysis in the case study

are immediately solved by the theorem prover. For example, the combination of constraint c_2 and method removeElement operate on different attributes. The proof is trivial and found by straightforward term rewriting using the Simplifier. Also trivial is the verification of *retrieval methods*, i.e. read-only methods such as the isConnectedTo methods. Their proofs are found requiring almost zero seconds of proof time. The real power of the theorem prover reveals itself in the cases where the constraint and method operate on the same attributes and/or persistent roots. For example, the combination of constraint c_4 and method removeElement takes 161.8 seconds on our test machine. The proof of this particular case involves many tedious steps, which deal with a combination of various object-oriented language features, such as late binding and down-casting. Also, the proof requires the assumption of another constraint in the schema: c_1 is necessarily assumed, since in order to extract the elements attribute from a CNode object, that object must be non-nil. Such additional assumptions are given as parameters to the `method_safety_goal` command.

Another observation that can be made is that object creation sometimes gives rise to considerably longer search times (e.g., method `CNode::createCNodeIn` with respect to constraint c_5). The reason for this is a technical one : in the goal, we need additional assumptions about the freshness of the new oid's, which leads to extra inference steps. Also, creation can involve the invocation of constructor methods.

7.4 Generation of feedback from unprovable subgoals

In the previous sections, we discussed the use of Isabelle to verify that the methods defined in a database schema do not violate the specified integrity constraints. The examples discussed so far are successful cases where the proof could be found automatically by Isabelle. One reason that the theorem prover fails to prove a goal is that the automated analysis technique is inherently limited in scope (see Section 7.2). Isabelle returns any subgoals that cannot be proved automatically. These goals should typically be examined further using Isabelle's interactive proof mode (although this requires a detailed knowledge of the theorem prover). Another possibility is that the goal we attempt to verify is in fact *false*; i.e., the method does not preserve the invariant because an error was made in the code. In this case, we would like the system to generate feedback, which makes the implications of the error clear enough to illuminates the error. In [SMS87], several different types of feedback are identified, including the derivation of weakest and sufficient preconditions. Below, we give directions how these forms of feedback are obtained as a by-product of our verification strategy.

7.4.1 Weakest preconditions

In transaction verification, one assumes that the database is in a consistent state before the transaction is executed, and attempt to prove that the constraint still holds afterwards. If the proof fails, the method can be corrected by adding a test which validates the constraint at run-time, *after* the transaction has been executed. This, however, may lead to expensive rollback operations. In general, a much simpler test can be obtained that can be evaluated *before* the transaction is executed. A test that is necessary and sufficient is called a *weakest precondition* [Dij76]. Given a transaction T and a constraint α, β is called a weakest precondition for α w.r.t. T, if for every database state D:

$$D \vdash \beta \text{ if-and-only-if } T(D) \vdash \alpha$$

In other words, a weakest precondition is a condition β that holds *before* the transaction T is executed if-and-only-if the constraint holds *afterwards*.

The concept of weakest preconditions is central to many existing techniques for fast run-time integrity checking (see Section 7.1.1). In this case, β is tested before the update is applied. If the test fails the update is rejected.

Along similar lines, one could also think of using weakest preconditions in conceptual design of update operations. In this case, the test β is used to actually correct the update such that it will always preserve the constraints. For this purpose we are interested in the class of weakest preconditions in OQL, for we could then use the condition to transform the transaction to:

$$\text{if } \beta \text{ then } T \text{ else skip}$$

which will maintain consistency while avoiding any rollback operations.

Consider the following definition of a method on the (abstract) class Node:

```
void addIncomingLinkWRONG(Link k) { incomingLinks += set(k) };
```

and the integrity constraint:

```
forall n in anodes : forall e in n.incomingLinks : e!=nil;
```

The automated proof procedure fails to verify that the constraint is an invariant of the method. The normalization step rewrites the initial term in the conclusion to the simplified updated constraint:

```
Level 1
1. (eval (os this) isNode)∧
    (∀n ∈ anodes | ∀e ∈ (get (os n) insOf) | eval (os e) isLink) →
     (∃n ∈ anodes | e = this ∧ this ∈ oids(os)) →
            (∀e ∈ (get (os this) (insOf ∘ addLink)) | eval (os n) isLink)
```

which is a weakest precondition for the method, *but at the level of its HOL representation*. We could try to define an OASIS-OQL formula that generates similar (at least equivalent) HOL code, but this leads to a rather complex condition that a programmer would normally not wish to put in his code, if only for reasons of efficiency. One of the problems is that weakest preconditions tend to be large disjunctions of formulae, in fact listing many alternative preconditions, some of which might be of interest to the programmer, whereas others might be not.

7.4.2 Sufficient preconditions

A weakest precondition provides a necessary and sufficient test for the method, which is a liberal requirement[1]. In [SMS87] it is argued that it may be cheaper in terms of run-time testing by only looking at a *sufficient precondition*; if true, this condition implies the weakest precondition. Observe that sufficient conditions include trivial tests, such as $false$. The task of the feedback generator is to generate meaningful alternative tests, rather than trivial ones. Based on the alternatives that are presented, the designer should then decide how to correct his code, e.g. by adding extra tests to the method, or by adding missing updates.

In [SMS87], candidate sufficient preconditions are generated from the weakest precondition, using a process that is called back-chaining. In our system, sufficient preconditions are obtained from the goals that are left by the safety verifier (i.e. after all three steps have been tried).

The example below illustrates the generation of sufficient preconditions. The same example is used as in the previous section. The automated analysis technique returns one unprovable subgoal:

```
Level 2
1.!! name pos inLinks outs content
   [| ∀n ∈ anodes | ∀e ∈ (get (os n) inLinksOf) | eval (os e) isLink;
     os this = Some(ANode name pos inLinks outs content);
     this ∈ anodes |] ⇒ eval (os k) isLink
```

The conclusion of this goal provides a sufficient precondition for the method to preserve the constraint: the conclusion corresponds to the OASIS-OQL formula ‘k instanceof Link’, which is the same as the condition ‘k != nil’. Alternative tests are obtained from the goal by negating assumptions, e.g. the OASIS-OQL formula ‘not(this in anodes)’, which is obtained from the negation of the third assumption ‘this ∈ anodes’ in the goal. The negation of the other assumptions in the goal does not lead to a very meaningful tests: the first assumption represents the constraint itself (which is assumed to hold in an arbitrary state) and the

[1]The weakest precondition is often called the *weakest liberal precondition* [2].

second condition is a *typing assertion*, which restricts the dynamic type of the this-object to be ANode. Such typing assertions occur frequently during proofs, as they result from splitting invocations of the `eval`-predicate as discussed in Section 7.2.3. Hence, from the above goal we obtain two possible OQL preconditions for the method addIncomingLinkWRONG:

1. not(this in anodes)

2. k instanceof Link

From these conditions, numerous options are available to proceed method correction. For example, the method could be corrected by including one of the tests being generated above (or both), as a precondition in the code (Indeed, there are numerous other possibilities: *any* condition that implies these tests is sufficient). Another possibility, suggested in [SMS87], is that sometimes the method should be corrected by adding extra updates, rather than tests. Thus, from the conditions that are generated, we can only express our hope that the schema designer will realize what kind of corrections should be made to the code. After all, it is up to the designer to decide which of the available solutions is most apt.

7.4.3 Improving feedback generation

The method presented for the derivation of sufficient preconditions should eventually be supported by tools. In our view, an ideal feedback generator should provide facilities to extract candidate preconditions from the unproved subgoals and present these to the designer. This has only been partly realized thus far.

One of the difficulties in feedback generation is that reasoning takes place at the semantic level (HOL), rather than at the conceptual level (OO). Although in principal it is possible to train consultants such that they can interpret the sometimes rather cryptic HOL code, ideally feedback is presented in a more "digested" form.

Feedback generation can be improved significantly by "polishing" the low semantic information in the goal. For example, OQL notation can be used for operations on sets and quantifications, rather than HOL notation. Similarly, invocations of '`get`', which involve large case splits, can be transformed back to simple attribute selections (i.e., the inverse of the mapping shown in Section 5.7). The same applies to the '`eval`' predicate, which corresponds to the instanceof predicate in OQL. Using so-called *parse translations*, such syntax transformation can be implemented easily in Isabelle. Parse translations are functions on the abstract syntax tree, which determine how the formulae appear at the output. For example, the theory of objects includes a parse translation for invocations of `eval` , which "computes" the least upper bound of the cases that are marked *true*, based on the inheritance relation in the current OASIS schema. Also invocations of `get` are printed using normal dot-notation, thus hiding the actual case-splits underneath.

7.5 Discussion of related work

7.5.1 Integrity checking

Several authors have previously addressed the problem of efficient integrity checking in the context of either relational databases (e.g. [Nic82, HI85, Qia88]), deductive databases (e.g. [BDM93, Sel95]), or object-oriented databases [Law95]. As pointed out in [BGL96], a common idea underlying these methods is that efficiency improvement is obtained from the assumption that the constraint holds prior to the update. Furthermore, the goal is to derive a test that can be evaluated before the update is applied, thus avoiding expensive rollback operations. Such a test that is necessary and sufficient is called a *weakest precondition* [Dij76]. In the case of deductive databases, the problem is complicated by the fact that an update may also induce the introduction or removal of derived data.

In the context of relational databases, Hsu and Imielinski ([HI85]) present a method for efficient integrity checking. Constraints should be in prenex normal form, but updates may affect several relations. A so-called "updated constraint" is derived for each constraint-update pair, which effectively provides a weakest precondition for the update. The paper presents a simplification algorithm to transform an updated constraint to an AND-OR combination of simpler constraints. The algorithm takes advantage of the assumption that the constraint holds prior to the update.

Also in the context of of relational databases, Qian [Qia88] presents an effective method for integrity constraint checking. Similar to [HI85], an algorithm is presented for deriving a weakest precondition, which is then simplified under the assumption that the constraint holds prior to the update. Constraints are more general than in [HI85]: there is no restriction on the number of variables ranging over a relation and constraints do not need to be in prenex normal form. It is claimed that more simplifications are obtained than using the algorithm of [HI85], but this claim is not motivated.

Lawley [Law95] defines weakest preconditions for an object-oriented database programming language, without iteration. An algorithm is presented for deriving the weakest precondition for a given update-constraint pair. As also pointed out in [Qia88, HI85, LT93, BGL96], it is argued that this condition may be further simplified under the assumption that the integrity constraint holds before the update is applied. No simplification algorithm is presented. Instead, the author suggests that simplification can be done using existing techniques for semantic query optimization: the condition is then observed as a boolean-valued query which needs to be transformed to a more efficient form. It is not shown, however, how we could benefit from such techniques.

Our integrity verification algorithm subsumes the above existing ideas for the simplification of integrity checking. As part of the verification algorithm, an updated constraint is derived which is then further simplified under the assumption that the constraint holds prior to the update. A major difference with the previous work is that our simplification algorithm is implemented (namely using Isabelle's Simplifier tool) and that it exploits several of the tool's advanced features for term-rewriting, e.g. congruence rules and local assumptions We have demonstrated the additional simplifications that can be obtained using these techniques.

Our method supplements the local processing technique of Jagadish and Qian ([JQ92]), who present an algorithm for efficient run-time integrity checking of constraints in object-oriented databases. They show how declarative global specifications of constraints can be transformed to a form that permits efficient local processing. Our method can be used to identify the case for which integrity is always preserved and thus no tests need to be performed at all.

7.5.2 Integrity verification.

Our work follows the line of research set out by Sheard & Stemple ([SS89]) on the automated verification of database transaction safety. In [SS89] a system is presented to prove at compile-time that a transaction can never violate the integrity constraints of the database. The transaction safety verifier component is implemented using *term-rewriting*. The term-rewriter uses a large knowledge base, which stores general knowledge about the transaction and constraint language. This includes basic theorems, such as a rule asserting the commutativity of the set-union operation. Much of the power of the Sheard & Stemple system derives from adding more problem-specific rules, which enable the simplification of terms that frequently appear during transaction safety analysis. Our approach using Isabelle/HOL differs in that it uses the object-oriented rather than the relational framework. Also, our safety verifier uses *natural deduction* in addition to *term-rewriting*. Initially, we have tried to follow the approach of [SS89], using Isabelle and its term-rewriter ("Simplifier") as a platform of implementation. In our experiments, however, it was not clear how goals could be solved entirely by term-rewriting. Goals can take many different forms and it is not clear to us how these can be proved *systematically* by term-rewriting. We soon ended up adding many new non-standard rewrite-rules to the Simplifier. Quite often, it was doubtful if these rules were relevant in the context of other database specifications as well [SB97]. By including natural deduction in the safety verifier, we have eliminated the need for adding such rules. In fact, our safety verifier is Isabelle's general strategy for automated theorem proving in HOL, with basic extensions for our new theory of objects.

The approach of [BD95, BS96, BS97, BS98] is based on an object-oriented database programming language. A framework is presented for the compile-time verification of a method preserving a number of static integrity constraints. The language that is studied by the authors is based on the database programming language O2C [BDK92], which is comparable to the OASIS language. Their analysis starts with a simple compilation technique to identify those combinations of transaction and constraint that are certainly not in conflict because the transaction and constraint access different attributes or persistent roots. In our system, this analysis is implemented using term-rewriting (see the normalization step (ii) of the automated verification strategy). For those combinations of transaction and constraint that could not be proved safe in the first step, Benzaken *et al* use a second more detailed analysis. This analysis is a variant of Dijkstra's classical notion of predicate transformer, as mentioned above. Two alternative notions of predicate transformer are introduced, i.e. a so-called *forward* and *backward* predicate transformer. Using the backward predicate transformer, transaction safety is verified as follows. In transaction safety, we have to prove an implication: we assume that the constraint holds in an arbitrary state, and we need to prove that the constraint is still valid after the transaction is executed. In abstract let us denote this implication as $P \Rightarrow Q$. Using the *backward* predicate transformer, another formula R is extracted from the transaction and constraint code. This formula only provides a *sufficient* not necessary test for Q; i.e. $R \Rightarrow Q$. The transaction is now proved safe with respect to the constraint, by verifying the implication $P \Rightarrow R$, instead of $P \Rightarrow Q$ (indeed, the latter condition is a direct consequence of the former). The verification is carried out using a first-order logic based automated theorem prover, although no details are given how the theorem prover is actually used. A similar strategy is used for the forward predicate transformer, which also generates a sufficient test for a method.

Their alternative predicate transformers are aptly called an *abstract interpretation*: since some information is omitted by only looking at sufficient tests, no real semantics is provided for the transaction. This is in sharp contrast to Dijkstra's predicate transformer, which might be viewed as an alternative way of defining the semantics, as pointed out in [Win93]. The claim is that their forward and backward predicate transformers provide "some useful and correct information about the run-time behavior of the program", but it might very well be that in the end safety cannot be proved by the theorem prover simply because the condition was incomplete. This problem does not arise in our analysis technique, although an automatic proof may fail because the technique is inherently incomplete (see Section 7.2). In this case, human interaction is needed to complete the proof. However, the invariants in the case study could be verified entirely by automated reasoning.

A more serious problem with the approach is *soundness*: many intricate object-oriented features are abstracted away by their predicate transformers and it is not clear as to whether the definitions are correct or not. As an example, consider the following constraint and persistent root declaration (taken from [BD95]):

```
name Persons: set (Person);
forall p in Persons: p.spouse.spouse=p or p.spouse=nil;
```

As seen in the code, the approach of Benzaken *et al* accommodates nil values. And indeed, the O2 system allows nil-references to be inserted into the set Persons [BDK92]. It is unclear to us how nil-references are dealt with in their work (e.g., consider the case where p=nil in the above constraint). However, in [BD95] several methods are proved safe with respect to the above constraint.

Chapter 8

Verification support for semantic-based transaction models

Verification support for semantics-based transaction models, based on the invariant analysis technique discussed in Chapter 7, is studied by examining a cases study for a particular model, namely the CoAct model. Results include experiments for the verification of user-defined compensating methods (i.e., the model requires the user to define for each method m, another method m^{-1} that can be invoked to "undo" the effects of an invocation of m); verification using the Isabelle is shown to be feasible.

8.1 Semantics-based transaction models

Transaction models provide a mechanism to manage the concurrent access of shared data, thus relieving the user from concurrency control problems. The basic transaction model is the so-called *read/write model* [EGLT76, VWP$^+$97]. In the read/write model, the four fundamental properties of database transactions, i.e., *atomicity*, *consistency*, *isolation*, and *durability* (ACID) are enforced based on a registration of the low-level read and write operations performed on data items. However, the basic model is inadequate for advanced applications of database technology, such as cooperative work and workflow applications [Elm92, Özsu94, VWP$^+$97]. For this reason, so-called *semantics-based transaction models* have been developed in recent years. The idea here is that additional information is exploited about the semantics of high-level data operations , rather than examining only the low-level decompositions of these operations into sequences of reads and writes. We consider the case of an object-oriented database, where the high-level data operations are methods operating on persistent objects.

One particular semantics-based transaction management idea is the use of *compensating transactions*, where for each operation, an "inverse" (or "undo") operation has to be provided (e.g., see [CD97, MR97]). The intension is that a compensation operation semantically undoes the effects of the original operation—it does not merely restore a previous state. Although the correctness of compensation operations is often assumed, little attention has been devoted to the actual definition and verification of these operations.

In the early nineties, Korth *et al* presented a formal approach to recovery by compensating transactions [KLS90]. Their ideas have been incorporated in numerous transaction models since then. In [KLS90], three guidelines are given for the specification of compensating transactions. The first of these guidelines, so-called 'Constraint 1', is of interest to us here. Informally, this constraint asserts that if a transaction T is immediately followed by its the compensating transaction CT, then the composed transaction '*T followed by CT*' should be the identity transaction. Our goal is to develop tool support for the verification of 'Constraint 1' for given T and CT pairs.

Another concept used in semantics-based transaction models is the notion of commutativity (also called compatibility). Informally, commutativity of two operations means that the operations do not conflict with each other. The idea is that by identifying cases where the operations do not conflict, the system can offer more concurrency. The schema designer is required to specify so-called commutativity relations, where for each method pair conditions on the actual parameter values are given. A goal of our work is to develop tool support to verify that the specified conditions are sufficient to guarantee that the operations do not conflict.

The chapter is organized as follows. In Section 8.2, we consider compensation and commutativity in the context of a particular semantics-based transaction model, namely the CoAct model [RKT$^+$95, KTWK97]. Section 8.3 then shows how to define compensating methods. Compensating methods are given for the SEPIA schema in Appendix A. Section 8.4 discusses the use of the Isabelle system for verifying the correctness of these methods. In Section 8.5, an example of verifying the commutativity of a method pair is shown. The example illustrates that the verification approach has also potential for this kind of analysis. Section 8.6 discusses the intricate issues that arise in the verification of compensation due to object creation. In Section 8.7, we give an overview of the experimental results. The experiments demonstrate the feasibility of the approach. Finally, Section 8.8 gives conclusions and discusses future work.

8.2 Compensation and commutativity in the CoAct model

The CoAct model [WK96, RKT$^+$95, KTWK97] is a semantics-based transaction model used to support cooperative applications. The model makes use of three semantics-based transaction management ideas: *compensation* [KLS90], *backward commutativity* [Wei88, Wei93], and *forward commutativity* [Wei88, Wei93, LMWF94].

In the CoAct model, users have the possibility to selectively undo the effects of previously performed operations in order to integrate their work with that of other users. A *save point mechanism* does not provide adequate support for undoing work in a cooperative work environment: restoring a saved state to undo an operation may have the side effect of undoing other operations (by other users). A compensating (or undo) operation provides a means to *exclusively* undo the effects of a previously performed operation. Conceptually, it is executed immediately after the operation it compensates. If interim operations have been executed after the to-be-compensated operation, these operations must commute right-backwards with (i.e., be independent of) it. Right-backward commutativity means that the interim operations can be moved "backwards" in the history, ahead of the to-be-compensated operation, with the result that the undo operation is executed immediately after it. This is illustrated in Figure 8.1: the to be compensated operation b_2 can be incrementally moved ahead in the history, to precede its compensating operation b_2^{-1}, provided that b_3 and b_4 are right-backward commutative with b_2. Both b_2 and b_2^{-1} can now be discarded from the history, provided that b_2

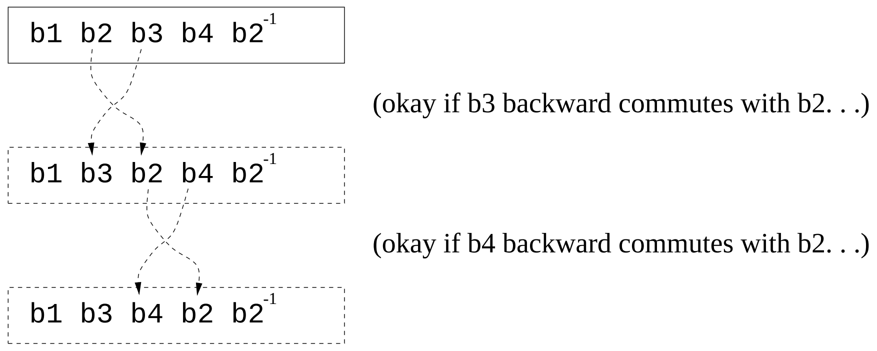
b1 b2 b3 b4 b2-1
(okay if b3 backward commutes with b2. . .)
b1 b3 b2 b4 b2-1
(okay if b4 backward commutes with b2. . .)
b1 b3 b4 b2 b2-1

Figure 8.1: History-based Compensation

followed by b_2^{-1} is indeed the identity operation. Backward commutativity ensures that the results of the intervening operations do not depend on the results of the to be compensated operation. The CoAct model also uses a slightly different notion of commutativity, namely *forward commutativity* [Wei88], which we will not discuss any further.

For the definition of the compensating operations and the commutativity relations, the CoAct model resorts to the assumption that this is done explicitly by the designer of the system. A compensating operation o^{-1} should be defined for each operation o; well-defined rules determine how the actual input parameters of o^{-1} are derived from the input parameters and the return value of o [EFK+96]. An example is discussed in Section 8.3. Also, the conflict (i.e., commutativity) relation between operations is defined by specifying simple conditions on the input parameters and return values of the operations. An example is discussed in Section 8.5.

We emphasize that the use of compensating operations and commutativity relations is not specific to the CoAct model. In fact, the ideas are far more general and more broadly applicable than sketched above. For an overview, the reader is referred to [VWP+97].

8.3 Definition of compensating methods

The OASIS database programming language does not provide an explicit mechanism for the definition of compensating methods (e.g., as part of the method body). Instead, we assume that for each method `M` in the schema, a compensating method `UNDO_M` is defined. The signature of `UNDO_M` is *derived* from the signature of `M`: the parameters are the same as for method `M`, plus an additional parameter of which the type corresponds to the output type of `M`. If the effects of `M` are to be compensated, `UNDO_M` is invoked with the same parameter values (and receiver), plus the return value of `M`. Figure 8.2 gives definitions of compensating methods for the methods defined in Appendix A. The definition of one of these methods is discussed below. Consider the definition of the removeElement method in the class CNode:

```
void Element::UNDO˙changeNameTo(string s, string rtn) {
    name = rtn
};

void Node::UNDO˙addIncomingLink(Link k, boolean rtn) {
  if (rtn) then incomingLinks -= set(k)
  else skip
};

void Node::UNDO˙addOutgoingLink(Link k, boolean rtn) {
  if (rtn) then outgoingLinks -= set(k)
  else skip
};

void ANode::UNDO˙removeAtomicContent(AtomicContents ac, ANode rtn) {
  if (rtn == this) then content += set(ac)
  else skip
};

void CNode::UNDO˙createANodeIn(string s, int p, ANode rtn) {
  elements -= set(rtn);
  anodes -= set(rtn)
};

void CNode::UNDO˙createLinkIn(Node a, Node b, string s, int p, Link rtn) {
  if (rtn!=nil) then {
      elements -= set(rtn);
      links -= set(rtn) }
  else skip
};

void CNode::UNDO˙removeElement(Element n, boolean rtn) {
  if (rtn) then elements += set(n)
  else skip;

void CNode::UNDO˙createCNodeIn(int p, int z, string s, CNode rtn) {
  elements -= set(rtn);
  cnodes -= set(rtn)
};
```

Figure 8.2: Example (UNDO) Operations for the SEPIA Schema

```
boolean CNode::removeElement(Element n) {
  if (n != nil) and (n in elements) and
      (forall x in elements : not(x.isConnectedTo(n))) then {
        elements -= set(n); return true
  } else return false
};
```

The method removeElement removes one of the elements (i.e. Node or Link) from the elements of a given CNode object, provided that that element occurs and there are no other connections (within the same CNode object) to that element. The method returns *true* if the condition is met (indicating that the Element object has been successfully removed); otherwise *false* is returned. A method that compensates the effects of the above method could be defined as follows:

```
void CNode::UNDO_removeElement(Element n, boolean rtn) {
  if (rtn) then {
     elements += set(n)
  } else skip
};
```

Notice the use of the extra parameter rtn for the return value; often this parameter is useful to define the compensating method.

The definition of the compensating method gives rise to a proof obligation. For the above example, it has to be verified that for any database state DB, receiver object o, and input parameter values n, it is the case that o.removeElement(n) executed in the state DB, immediately followed by o.UNDO_removeElement(n, r) results in a state DB' that is equivalent to DB (where r is the value returned by the removeElement method). Using the Isabelle theorem prover, we have verified this theorem automatically in 8.7 seconds on a workstation. Details of how the proof is performed are discussed in the next section.

8.4 Compensation analysis using Isabelle

The definition of a compensation method gives rise to a *proof obligation*, which can be verified using Isabelle. Consider a method M with n input parameters. Informally, we have to prove that for an arbitrary database state DB, receiver object o, and input parameter values $v_1 \cdots v_n$, it is the case that $o.\mathtt{M}(v_1, \ldots, v_n)$ executed in state DB, immediately followed by $o.\mathtt{UNDO\_M}(v_1, \ldots, v_n, r)$ results in the same old state DB, where r is the value returned by M. Notice that the proof obligation involves a universal quantification over all possible database states, receiver object, and parameter values. Based on the representation of methods in HOL, such a requirement can be entered as a proof goal in Isabelle.

The proof of the compensation theorems is a difficult task, which we attempt to automate as much as possible using the Isabelle theorem prover. Our analysis technique is based on Isabelle's default machinery for automated theorem proving in HOL. The default machinery of the theorem prover uses a combination of term-rewriting (Isabelle's Simplifier) and natural deduction (Isabelle's Classical Reasoner). In this section, we describe how to use these techniques and tools for compensation analysis. This involves the following steps:

1. formulation of the initial proof goal
2. normalisation of the goal
3. depth-first proof search

These steps are carried out successively, without human interaction. If the automated proof procedure does not find a proof, it returns the goals it can not solve. These goals can then be analysed further, using Isabelle's interactive proof mode [Isa].

8.4.1 Step 1: Formulation of the proof goal

In our formal framework, verifying the correctness of the implementation of a compensation method amounts to proving a formal theorem in HOL, which we refer to as a *compensation theorem*. This theorem asserts the correctness of a compensation method m^{-1} with respect to the implementation of the corresponding "forward" method m. Such theorems are defined in terms of the functional HOL representations of the methods m^{-1} and m. The theorem is entered as a proof goal to the Isabelle system.

We construct a proof goal for the database schema by specifying boolean-valued expressions about method applications with respect to the input object store, the persistent roots, and the parameter values. To express what we mean by a compensation requirement for method `UNDO_M` with respect to method `M`, consider the following applications (and results) of the two methods:

$$\text{let } (\Delta, v_1', \ldots, v_n', r) = \texttt{M}\ os\ v_1\ \cdots\ v_n$$
$$\text{let } (\Delta', v_1'', \ldots, v_n'', r') = \texttt{UNDO\_M}\ (\texttt{smash}\ os\ \Delta)\ v_1\ \cdots\ v_n\ r$$

The UNDO method is applied to the same arguments as the original method, with the exception of the object store argument; it is also applied to the return value of `M`. The delta value Δ contains the (tentative) changes to the database state made by method `M`. These changes are combined with the original object store in order to evaluate method `UNDO_M`. The operator `smash` is used to hide (override) any bindings for the same objects in its first argument (i.e., it applies the changes to the database state). The above notation omits changes to the persistent roots. For our example schema, values v_1, v_2, and v_3 are actually case expressions that check for changes to the persistent roots; they pass on either the initial value or the modified value.

As can be seen above, method `UNDO_M` is applied to an `M`-modified object store; it returns a delta value Δ' that contains its (tentative) changes to the database state. For compensation, we want to prove that the original object store os is equivalent in some sense to the object store obtained after the changes proposed by both methods are applied (i.e., we want to show $os \equiv \texttt{smash}\ (\texttt{smash}\ os\ \Delta)\ \Delta'$).

A compensation proof goal is universally quantified over the *defined* objects in the initial object store. For these objects, Isabelle reasons about the equivalence of each object's value in the original and updated object stores. The equivalence is not equality. The initial proof goals that must be entered to the Isabelle system are quite large. We have defined an ML function to automate their generation for a particular schema. To do this, the user types in a command such as the following:

```
- start_proof(undo_method_goal("CNode","removeNodeOrLink"));
```

The generated goal corresponds to the let-expression form shown above, but includes additional parameters for the persistent roots. It also includes an assumption about the "definedness" of nested object identifiers in the initial object store, and assumptions about the "freshness" of any new oids needed for object creation. The generated goal is shown below.

```
Level 0
(eval (os this) isCNode) ∧ (x ∈ oids os) →
 let (Δ, cnodes₁, links₁, anodes₁, this₁, n₁, rtn₁) =
          CNode_removeElement os cnodes links anodes this n;
      (Δ⁻¹, cnodes₂, links₂, anodes₂, this₂, n₂, rtn₂) =
          CNode_UNDO_removeElement
               (smash os Δ)
               (case cnodes₁ of None ⇒ cnodes | Some y ⇒ y)
               (case links₁ of None ⇒ links | Some y ⇒ y)
               (case anodes₁ of None ⇒ anodes | Some y ⇒ y)
               this n rtn₁
 in (get ((smash os (smash Δ Δ⁻¹)) x) idf = get (os x) idf)∧
    (case cnodes₂ of None ⇒ (case cnodes₁ of None ⇒ True | Some y ⇒ False)
                     | Some y ⇒ y = cnodes)∧
    (case links₂ of None ⇒ (case links₁ of None ⇒ True | Some y ⇒ False)
                    | Some y ⇒ y = links)∧
    (case anodes₂ of None ⇒ (case anodes₁ of None ⇒ True | Some y ⇒ False)
                     | Some y ⇒ y = anodes)
```

The goal examines an equivalence of states, comparing the object stores and persistent roots before and after the application of an update and its compensation. The equivalence of the object stores is asserted by the subformula:

$$\texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ \Delta\ \Delta^{-1}))\ \texttt{x})\ idf = \texttt{get}\ (\texttt{os x})\ idf$$

where Δ represents the update and Δ^{-1} its compensation. In other words, any lookup of the value of an object with oid '`x`' in the modified store '`smash os` $(\texttt{smash}\ \Delta\ \Delta^{-1})$' should result in the same value as the value obtained by a lookup of that object in the initial store '`os`'. The equivalence of the persistent roots, before and after the updates, is asserted by case statements.

8.4.2 Step 2: Normalization of the goal

Our analysis technique starts by applying simplifications to the initial proof goal, using term-rewriting. The main purpose of the rewritings is to apply simplifications to the subterm:

$$\texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ \Delta\ \Delta^{-1}))\ \texttt{x})\ \mathit{idf} = \texttt{get}\ (\texttt{os x})\ \mathit{idf}$$

in the goal, such that we obtain a normal form. The normal form embeds the updates (Δ's) in the lookup operation (`get`) such that we obtain a equivalent test, expressed in terms of initial state (`os`). We will call such a test an *updated test*.

The derivation of the updated test is implemented using the Isabelle Simplifier. The Simplifier performs term-rewriting with a set of theorems of the following form:

$$[|H_1\,;\ \cdots\ ;\ H_n|] \Rightarrow LHS = RHS$$

Such theorems are read as conditional rewrite rules: a term unifying with the expression on the left-hand side of the equality sign (LHS) is rewritten to the term that appears on the right-hand side (RHS), provided that the hypotheses $(H_1, \ldots, H_n)$ hold. The default Isabelle Simplifier installs a large collection of standard reduction rules for HOL; new rules can be added to customize the Simplifier to particular tasks. At present, 49 new rewrite rules have been added to support the theory of objects. There are rules for simplifying delta patterns (shown in the Figures 5.2,5.3,5.4,5.5). The rules shown in Figure 5.5 are used to derive the updated test; using these rules, a pattern such as $\texttt{get}\ ((\texttt{smash os}\ \Delta))\ x)\ f$ can be reduced to an equivalent expression in terms of the initial state `os` (i.e., without Δ's). For our example goal, the following goal is obtained via rewriting:

```
Level 1
eval (os this) $isCNode$ ∧ x ∈ oids(os) →
k ∈ get (os this) $eltsOf$ ∧ x = this →
      get (os this) ($addElt$ ∘ $delElt$) = get (os this) $idf$
```

The Simplifier performs many complex rewriting steps behind the scenes. This involves the rules in Figures 5.2 and 5.5, as well as the standard rewrite rules for HOL. Below we illustrate the derivation of the updated test using these rules.

Consider the goal that is generated for the removeElement method. The actual Δ and Δ^{-1} values plus the return value of the method are obtained by first expanding the method definitions that are generated by our schema translator. The definition of the removeElement is found on page 51; in the simplified representation below, we do not show the persistent roots and the return value of the undo method:

$$\begin{array}{rcl} \Delta & \equiv & \texttt{if}\ \mathit{cond}\ \texttt{then set os this}\ \mathit{delElt}\ \texttt{else skip} \\ \Delta^{-1} & \equiv & \texttt{if rtn then (set (smash os}\ \Delta\texttt{) this}\ \mathit{addElt}\texttt{) else skip} \end{array}$$

where $\texttt{rtn} \equiv (\texttt{if}\ \mathit{cond}\ \texttt{then true else false})$. Notice that the Δ^{-1} takes the modified object store $(\texttt{smash os}\ \Delta)$. Isabelle derives the updated test by applying simplifications to the formula:

$$\texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ \Delta\ \Delta^{-1}))\ \texttt{x})\ \texttt{idf} = \texttt{get}\ (\texttt{os x})\ \mathit{idf}$$

that appears in the conclusion of the goal, using the rewrite rules for objects. The simplification is performed using several advanced features of the Isabelle Simplifier. Simplification starts by expanding the let-definitions. This results in a substitution of the above definitions of Δ and Δ^{-1}:

$$\begin{array}{l}
\texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ (\texttt{if}\ \mathit{cond}\ \texttt{then}\ (\texttt{set os this}\ \mathit{delElt})\ \texttt{else skip})\\
\qquad\qquad\qquad (\texttt{if}\ (\texttt{if}\ \mathit{cond}\ \texttt{then true else false})\\
\qquad\qquad\qquad\ \ \texttt{then}\ (\texttt{set}\ (\texttt{smash os}\ (\texttt{set os this}\ \mathit{delElt}))\ \texttt{this}\ \mathit{addElt})\\
\qquad\qquad\qquad\ \ \texttt{else skip})))\ \texttt{x})\ \mathit{idf} = \texttt{get}\ (\texttt{os x})\ \mathit{idf}
\end{array}$$

Various simplifications can be applied to this goal. The Simplifier first rewrites the conditional that is substituted for the return value in Δ^{-1} and combines the nested `set` operations using the rule labeled **[set_set]** in Figure 5.2:

$$\begin{array}{l}
\texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ (\texttt{if}\ \mathit{cond}\ \texttt{then}\ (\texttt{set os this}\ \mathit{delElt})\ \texttt{else skip})\\
\qquad\qquad\qquad (\texttt{if}\ \mathit{cond}\ \texttt{then}\ (\texttt{set os this}\ (\mathit{addElt} \circ \mathit{delElt}))\\
\qquad\qquad\qquad\ \ \texttt{else skip})))\ \texttt{x})\ \mathit{idf} = \texttt{get}\ (\texttt{os x})\ \mathit{idf}
\end{array}$$

Subsequently, Isabelle splits the conditional. This is one of the advanced features of the Simplifier. So-called *splitting rules* are used to reduce function applications [Pau94]. The splitting of conditionals is performed using the rule:

$$P(\texttt{if}\ Q\ \texttt{then}\ x\ \texttt{else}\ y) \equiv ((Q \to P(x)) \land (\neg Q \to P(y)))$$

The application of such rules is expensive and Isabelle only applies them after simplification. In the goal shown above, there are two conditionals. By splitting the first conditional, the Simplifier will rewrite the above goal to:

$$\begin{array}{l}
\mathit{cond} \to (\texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ (\texttt{set os this}\ \mathit{delElt})\\
\qquad\qquad\qquad (\texttt{if}\ \mathit{cond}\ \texttt{then}\ (\texttt{set os this}\ (\mathit{addElt} \circ \mathit{delElt}))\\
\qquad\qquad\qquad\ \ \texttt{else skip})))\ \texttt{x})\ \mathit{idf} = \texttt{get}\ (\texttt{os x})\ \mathit{idf}) \land\\
\neg\mathit{cond} \to (\texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ (\texttt{skip})\\
\qquad\qquad\qquad (\texttt{if}\ \mathit{cond}\ \texttt{then}\ (\texttt{set os this}\ (\mathit{addElt} \circ \mathit{delElt}))\\
\qquad\qquad\qquad\ \ \texttt{else skip})))\ \texttt{x})\ \mathit{idf} = \texttt{get}\ (\texttt{os x})\ \mathit{idf})
\end{array}$$

After splitting, the Simplifier will try simplification once more. Rewriting continues using an advanced feature of the Simplifier called “local assumptions”. For instance, in simplifying the conclusion of an implication, the tool uses additional rewrite rules it extracts from the assumption. This is done using the congruence rule for implication:

$$[|\ P_1 \equiv Q_1\ ;\ Q_1 \Rightarrow P_2 \equiv Q_2\ |] \Rightarrow (P_1 \to P_2) \equiv (Q_1 \to Q_2)$$

Given this rule, the Simplifier assumes Q_1 (that is, the rewritten form of P_1) and extracts rewrite rules from it when simplifying P_2. In the example, the Simplifier extracts the rewrite rule $\mathit{cond} \equiv \texttt{true}$ for simplifying the conclusion of the implication on the left-hand side of the conjunction ($\land$) in the above formula: in this case, the Simplifier rewrites the if-part of the other conditional to *true*, such that the then-part is eliminated. The not-case is handled in a similar fashion. We obtain the following simplified formula:

$$\begin{array}{l} cond \rightarrow \\ \quad \texttt{get}\ ((\texttt{smash os}\ (\texttt{smash}\ (\texttt{set os this}\ delElt) \\ \qquad\qquad\qquad\qquad (\texttt{set os this}\ (addElt \circ delElt)))) \ \texttt{x})\ idf = \texttt{get}\ (\texttt{os x})\ idf\ \wedge \\ \neg cond \rightarrow \texttt{get}\ ((\texttt{smash os}\ (\texttt{smash skip skip}))\ \texttt{x})\ idf = \texttt{get}\ (\texttt{os x})\ idf \end{array}$$

The Δ-patterns are now further simplified, using the rule labeled **[smash_set_set]** and the rule labeled **[smash_right_id]** in Figure 5.2. The rule **smash_right_id** is applied twice. Now, the not-case is eliminated by the Simplifier using standard rewriting properties of $\wedge$, $=$, and $\rightarrow$; we obtain:

$$cond \rightarrow \texttt{get}\ ((\texttt{smash os}\ (\texttt{set os this}\ (addElt \circ delElt)))\ \texttt{x})\ idf = \texttt{get}\ (\texttt{os x})\ idf$$

Now, Isabelle eliminates the `set` operation (we also use `oids_set` for simplifying the resulting condition for the oid). This is done using the rule labeled **[get_set]** in Figure 5.2:

$$\begin{array}{l} cond \rightarrow (\texttt{if}\ (\texttt{x} \in \texttt{oids os}) \wedge \texttt{x} = \texttt{this then get}\ (\texttt{os x})\ idf \circ (addElt \circ delElt) \\ \qquad\quad \texttt{else get}\ (\texttt{os x})\ idf) = \texttt{get}\ (\texttt{os x})\ idf \end{array}$$

The Simplifier removes the condition '$\texttt{x} \in \texttt{oids os}$' since this condition was assumed to hold in the initial proof goal (i.e., at `Level` 0) and it removes the identity function in the function compositions. Furthermore, splitting of the conditional will remove the else-part. We arrive at the following simplified formula:

$$\begin{array}{l} cond \wedge \texttt{x} = \texttt{this} \rightarrow \\ \quad (\texttt{get}\ (\texttt{os this})\ (addElt \circ delElt)) = (\texttt{get}\ (\texttt{os this})\ idf) \end{array}$$

As is seen in the derivation above, standard rewriting techniques are used to simplify the initial compensation requirement. The rewriting steps are implemented using the Isabelle Simplifier, with extra rules installed for simplifying Δ-patterns.

8.4.3 Step 3: Depth-first proof search

In the previous section, we demonstrated the use of standard rewriting techniques for simplifying the initial proof goal. In most cases, however, the use of these techniques is not sufficient to completely solve the goal.

The automated verification strategy uses a combination of term-rewriting with deductive techniques. Isabelle's Classical Reasoner tool automates the reasoning with natural deduction calculi. The tool implements several "naive" search algorithms, such as a depth-first and a best-first search. The default configuration of the tool supports the standard data types: it handles quantifiers and rules are installed to reason about sets and lists. Extensions we have made add natural deduction style introduction and elimination properties for the theory of objects.

We use the tool's depth-first search algorithm. This handles substantial proofs: variables introduced by the use of quantifiers are automatically instantiated, and backtracking is performed to handle alternative unifiers. Between deduction steps, Isabelle invokes the Simplifier to attempt further rewriting. Hence, the analysis technique uses a tight integration of rewriting and deductive techniques. Using a depth-first search interleaved with term-rewriting steps, the rest of the proof goes automatically. The proof is found in 8.7 seconds, running Isabelle on an average workstation.

To illustrate the power of the theorem prover, and the use of standard deductive techniques, it is instructive to examine some details of the proof steps performed behind the scenes. Our starting point is the goal that is left after the application of the initial simplification step (Level 1). First, the Classical Reasoner recursively performs any possible *safe inferences* on this goal. Safe inferences are the ones that can be applied deterministically; they do not introduce unknown terms, or instantiate variables in the goal. Hence, there is no need to undo any of these steps at later stages in the proof. In case of the above goal, safe inferences apply to the logical connectives (that is, the standard rules $\rightarrow$-Introduction, and $\wedge$-Elimination—see Section 4.2.3) and the elimination of `eval` (see Figure 7.2.3).The result is a list of three subgoals:

```
Level 2
1.!! refDir showStmt URL.
   [| os this = Some(AtomicContents refDir showStmt URL); False;
      x ∈ oids(os); k ∈ get (os this) eltsOf;
      x = this |] ⇒ get (os this) (addElt ∘ delElt) = get (os this) idf
2.!! name pos ins outs content.
   [| os this = Some(ANode name pos ins outs content); False;
      x ∈ oids(os); k ∈ get (os this) eltsOf;
      x = this |] ⇒ get (os this) (addElt ∘ delElt) = get (os this) idf
3.!! name pos ins outs size elements.
   [| os this = Some(CNode name pos ins outs size elements); True;
      x ∈ oids(os); k ∈ get (os this) eltsOf;
      x = this |] ⇒ get (os this) (addElt ∘ delElt) = get (os this) idf
4.!! name pos from to.
   [| os this = Some(CNode name pos from to); False;
      x ∈ oids(os); k ∈ get (os this) eltsOf;
      x = this |] ⇒ get (os this) (addElt ∘ delElt) = get (os this) idf
```

There exists a subgoal for each type of the receiver object (`this`). By static typing, we know that only the second subgoal is relevant, namely the one for `CNode`. The other subgoals for `ANode` and `Link` denote wrongly typed cases (the receiver is known to be of static type CNode); the proof of these subgoals is trivial because *false* is included in their list of assumptions. Hence, only one subgoal is left:

```
Level 3
1.!! name pos ins outs size elements.
   [| os this = Some(CNode name pos ins outs size elements);
      x ∈ oids(os); k ∈ get (os this) eltsOf;
      x = this |] ⇒ get (os this) (addElt ∘ delElt) = get (os this) idf
```

At this stage, no further safe inferences apply. The Classical Reasoner now first calls the Simplifier, before any unsafe inferences are tried. The Simplifier rewrites the conclusion of the goal in three steps, as follows:

$$\begin{aligned}
&\texttt{get}\ (\texttt{os this})\ (\mathit{addElt} \circ \mathit{delElt}) = \texttt{get}\ (\texttt{os this})\ \mathit{idf} \\
\overset{(1)}{\equiv}\ &\texttt{get Some}(\texttt{CNode name pos ins outs size elements})\ (\mathit{addElt} \circ \mathit{delElt}) = \\
&\texttt{get Some}(\texttt{CNode name pos ins outs size elements})\ \mathit{idf} \\
\overset{(2)}{\equiv}\ &\texttt{CNode name pos ins outs size}\ ((\texttt{elements} - \{\texttt{k}\}) \cup \{\texttt{k}\}) = \\
&\texttt{CNode name pos ins outs size elements} \\
\overset{(3)}{\equiv}\ &((\texttt{elements} - \{\texttt{k}\}) \cup \{\texttt{k}\}) = \texttt{elements}
\end{aligned}$$

We briefly explain these steps. In step (1), Isabelle uses the head of the list of assumptions of the goal (i.e., the formula $\texttt{os this} = \texttt{Some}(\texttt{CNode} \cdots)$ as a rewrite rule. In step (2), the Simplifier applies the rule labeled **[get_Some]** in Figure 5.5. The resulting term is then further simplified in step (3) using a standard rewrite rule derived by Isabelle for any data type definition: two applications of the CNode constructor are the same if-and-only-if their arguments are equivalent. The following simplified goal is obtained:

```
Level 4
1.!! name pos ins outs size elements.
   [| os this = CNode name pos ins outs size elements; x ∈ oids(os);
      k ∈ elements;
      x = this |] ⇒ ((elements − {k}) ∪ {k}) = elements
```

Isabelle repeats the process of applying safe inferences and simplification until it reaches a fixed point, where no more safe inferences and/or simplifications apply. Our goal is solved in the second pass, using standard introduction and elimination properties of sets (details are not shown); automated reasoning about sets is one of the standard assets of the theorem prover. Isabelle returns with the following message to indicate that the proof has succeeded:

```
Level 5
No subgoals!
```

The analysis tool implements an *automated proof strategy*. This is essentially the same strategy we have previously used for verifying consistency requirements (i.e., that a method preserves a number of static integrity constraints), and on which we have reported elsewhere [SE99]. The proof strategy is based on the standard machinery provided by the Isabelle theorem prover, which uses a combination of term rewriting and natural deduction [Pau94]. The standard machinery of the theorem prover supports the common data types in programming languages (e.g., int, bool, list, set, tuple). As discussed in Chapter 5, these data types are used to represent similar data types of the OASIS database language. Thus, we get most of the automated reasoning for free. For example, the set membership operation in the database language is mapped to the predefined set membership operation in HOL; Isabelle already installs a number of theorems to reason about set membership.

8.5 Backward commutativity analysis using Isabelle

In the previous section, we discussed the automated verification of compensation using the Isabelle theorem prover. In our analysis, we assumed that the compensating method m^{-1} is executed immediately after the "forward" operation m. This, however, may not always be the case. Compensation becomes more difficult if further methods have been applied, between the invocation of m and m^{-1}. The invocation of m is called *compensatable* [KTWK97] if the intervening operations commute backwards with (i.e., do not depend on) the results of m, such that they can be moved ahead of the to be compensated operation m without affecting the results of m or any subsequent operation (see Figure 8.1).

Typically, the backward commutativity relation is defined by the designer of a system: for each method pair, a simple condition on the input parameters and/or the return values should be specified to determine when invocations of these methods depend on each other or not. In this section, we give an example of a backward commutativity specification for the case study, and discuss its verification using the Isabelle theorem prover. The theorems that need to be verified turn out to be more intricate than those for compensation, but the same verification strategy can be used.

Consider the definition of the following method to add a Link object the incomingLinks collection of a Node object:

```
boolean Node::addIncomingLink(Link k) {
  if k == nil or k in incomingLinks then
     { return false
  else { incomingLinks += set(k) ; return true }
};
```

A sufficient precondition for an invocation of this method to commute backwards with another invocation of the same method is that the Link parameters denote different objects. The correctness of this condition is proved by Isabelle requiring 99 seconds of proof time on our test machine.

The procedure for verifying such conditions follows the procedure for verifying compensation methods. First, the backward commutativity requirement is specified as a formal theorem in HOL. In Figure 8.3, it is shown how this is done for our example method pair. We consider an arbitrary input state `os`, Node objects `na` and `nb`, and Link objects `la` and `lb`. The let-part of the goal examines the output of na $\cdot$ addIncomingLink$($`la`$)$ followed by nb$\cdot$addIncomingLink$($`lb`$)$ and vice versa. Both cases should result in the same return values and final state (including the persistent roots), provided that the receivers are of the correct type and that the (user-) specified precondition $\texttt{la} \neq \texttt{lb}$ holds. Notice that, in general, the receiver of the second method may possibly be an object created by the first method; thus, the type condition for '`nb`' takes the modified store, rather than the input store (although in this case no new objects are created).

The automated verification of this theorem is performed using the procedure outlined in the previous section, for verifying compensation. The proof starts by applying rewritings to the subformula

$$(\texttt{get}\ ((\texttt{smash}\ \texttt{os}\ (\texttt{smash}\ \Delta_2\ \Delta_1))\ \texttt{x})\ \mathit{idf} = \texttt{get}\ ((\texttt{smash}\ \texttt{os}\ (\texttt{smash}\ \Delta_4\ \Delta_3))\ \texttt{x})\ \mathit{idf})$$

```
let ($\Delta_1$, cnodes$_1$, links$_1$, anodes$_1$, na$_1$, la$_1$, rtn$_1$) =
        Node_addIncomingLink os cnodes links anodes na la;
    ($\Delta_2$, cnodes$_2$, links$_2$, anodes$_2$, nb$_2$, lb$_2$, rtn$_2$) =
        Node_addIncomingLink (smash os $\Delta_1$)
                             (case cnodes$_1$ of None $\Rightarrow$ cnodes | Some y $\Rightarrow$ y)
                             (case links$_1$ of None $\Rightarrow$ links | Some y $\Rightarrow$ y)
                             (case anodes$_1$ of None $\Rightarrow$ anodes | Some y $\Rightarrow$ y)
                             (nb) (lb)
    ($\Delta_3$, cnodes$_3$, links$_3$, anodes$_3$, nb$_3$, lb$_3$, rtn$_3$) =
        Node_addIncomingLink os cnodes links anodes nb lb;
    ($\Delta_4$, cnodes$_4$, links$_4$, anodes$_4$, na$_4$, la$_4$, rtn$_4$) =
        Node_addIncomingLink (smash os $\Delta_3$)
                             (case cnodes$_3$ of None $\Rightarrow$ cnodes | Some y $\Rightarrow$ y)
                             (case links$_3$ of None $\Rightarrow$ links | Some y $\Rightarrow$ y)
                             (case anodes$_3$ of None $\Rightarrow$ anodes | Some y $\Rightarrow$ y)
                             (na) (la)
in (eval (os na) isNode) $\wedge$
   (eval ((smash os (smash $\Delta_1$ $\Delta_2$)) nb) isNode) $\wedge$
   (x $\in$ oids(smash $\Delta_1$ $\Delta_2$)) $\wedge$
   (la $\neq$ lb) $\rightarrow$
   (rtn$_1$ = rtn$_4$ $\wedge$ rtn$_2$ = rtn$_3$ $\wedge$
   (get ((smash os (smash $\Delta_2$ $\Delta_1$)) x) idf) =
   (get ((smash os (smash $\Delta_4$ $\Delta_3$)) x) idf) $\wedge$
   (case cnodes$_2$ of  None $\Rightarrow$ (case cnodes$_1$ of None $\Rightarrow$ cnodes | Some y $\Rightarrow$ y)
                     |  Some y $\Rightarrow$ y) =
   (case cnodes$_4$ of  None $\Rightarrow$ (case cnodes$_3$ of None $\Rightarrow$ cnodes | Some y $\Rightarrow$ y)
                     |  Some y $\Rightarrow$ y)$\wedge$
   (case links$_2$ of  None $\Rightarrow$ (case links$_1$ of None $\Rightarrow$ links | Some y $\Rightarrow$ y)
                    |  Some y $\Rightarrow$ y) =
   (case links$_4$ of  None $\Rightarrow$ (case links$_3$ of None $\Rightarrow$ links | Some y $\Rightarrow$ y)
                    |  Some y $\Rightarrow$ y)$\wedge$
   (case anodes$_2$ of  None $\Rightarrow$ (case anodes$_1$ of None $\Rightarrow$ anodes | Some y $\Rightarrow$ y)
                     |  Some y $\Rightarrow$ y) =
   (case anodes$_4$ of  None $\Rightarrow$ (case anodes$_3$ of None $\Rightarrow$ anodes | Some y $\Rightarrow$ y)
                     |  Some y $\Rightarrow$ y))
```

Figure 8.3: Example backward commutativity goal without object creation

in the conclusion. Again, the purpose of these rewritings is to embed the updates (Δ-values) in the lookup operation (get) such that we obtain an equivalent test in terms of the state 'os' before the updates. The rewriting steps are similar to those illustrated in Section 8.4.2, but in this case the proof is more difficult since we compare two updated states, rather than comparing an updated state with the initial state (as is done in compensation analysis). The rewritten goal is solved using the Classical Reasoner, only requiring safe inference steps.

8.6 The issue of object creation

The example discussed in the previous section does not cover object creation, which raises several additional intricate issues in compensation analysis. We illustrate these issues using the following example method:

```
Link CNode::createLinkIn(Node na, Node nb, string s, int p) {
    if (na in elements and nb in elements) then {
       Link n = new Link(s,p,na,nb);
       elements += set(n);
       links += set(n);
       return n
    } else return nil
};
```

The above method creates a new Link object and adds that object to the elements collection of the receiver. Furthermore, persistence is provided to the object by inserting the object into the links persistent root.

How do we define a compensating method for the above method? Since the OASIS language uses persistence by reachability, the language does not provide an explicit delete operation to undo the creation of an object. Instead, to "delete" an object, we have to break all references to it; the result of breaking the references is that the object becomes invisible to the application programs. For example, we may define the compensating method for the createLinkIn method as follows:

```
void CNode::UNDO˙createLinkIn(Node na, Node nb, string s, int p, Link rtn) {
   if (rtn != nil) then {
      elements -= set(rtn);
      links -= set(rtn)
   } else skip
};
```

Indeed, this method breaks the references established by the "forward" method, but this may not be sufficient. The createLinkIn method has also returned a handle to the new object, and subsequent operations (performed by the transaction surrounding the invocation of the method) may be dependent on this result. Such dependencies are detected by the transaction management support. This is done by examining the backward commutativity relation: an invocation of the createLinkIn method cannot be undone if any of the subsequent operations fails to commute backwards with it.

The compensation theorem for verifying the correctness of the implementation of an undo method (as shown in Section 8.4.1) remains the same in case new objects are created. We examine the equivalence of the states only for the objects that were already previously defined in the input state, thus not including the newly created ones. It is sufficient to check that the undo method carefully breaks down all the references that have been made (the creation of an object using new does not have any visible effects itself).

8.7 Experimental results

All compensation methods for the case study, as shown in Figure 8.2 could be proved automatically by Isabelle. Table 8.1 gives experimental results for verifying the compensating

CLASS	METHOD	COMPENSATION	PROOF TIME
`Element`	`changeNameTo`	`UNDO_changeNameTo`	5.25s.
`ANode`	`removeAtomicContent`	`UNDO_removeAtomicContent`	15.29s.
`Node`	`addIncomingLink`	`UNDO_addIncomingLink`	7.42s.
`Node`	`addOutgoingLink`	`UNDO_addOutgoingLink`	7.55s.
`CNode`	`createANodeIn`	`UNDO_createANodeIn`	81.72s.
`CNode`	`createCNodeIn`	`UNDO_createCNodeIn`	85.34s.
`CNode`	`createLinkIn`	`UNDO_createLinkIn`	107.25s.
`CNode`	`removeElement`	`UNDO_removeElement`	14.17s.

Table 8.1: Experimental Results for Method Compensation

methods of the example schema. The proof times are in seconds, with Isabelle running on a SUN 296 MHz Ultra-SPARC-II, under Solaris. The times indicate that the proof strategy is reasonably efficient (although Isabelle performs up to a few hundred proof steps per second).

8.8 Conclusions and future work

In this chapter we have discussed the use of a theorem prover, namely Isabelle, for the automated verification of user-specified compensating methods. The analysis technique is based on a semantic representation of methods in higher-order logic as discussed in Chapter 5 of this thesis, and builds on Isabelle's standard machinery for automated theorem proving in HOL. The same technique has also been used for verifying that a method preserves database integrity, as discussed in Chapter 7. The technique automates the verification of proof goals as much as possible. The practical experiments discussed in this chapter indicate that many interesting cases of compensation can be verified automatically. The theorem prover returns any subgoals it cannot prove automatically. These may be examined further in Isabelle's interactive proof mode.

Commutativity analysis can in principle be carried out within the same formal framework, but the proofs turn out to be far more difficult. In fact, the commutativity problem can be viewed as a generalization of the compensation problem: in compensation analysis we verify that the sequential composition of two methods (i.e., the invocation of the "forward" method followed by the invocation of its undo) is equivalent to the state before the updates, whereas commutativity analysis involves the comparison of two sequences of method calls (i.e., the execution in the one order versus the other). We have briefly discussed one example of (backward) commutativity analysis, but further experiments are needed.

We do not know of any other (semi-) automated techniques for the analysis of compensating methods. Ammann *et al* [AJR97] apply formal methods to the semantic-based decomposition of transactions. In their work, the Z specification language ([Spi92]) is used to formally define transactions. They focus on decomposing a transaction into steps that preserve certain properties, including database integrity constraints and a compensation property. The analyses and proofs in [AJR97] are done by hand, for the specific example schema used in the paper. The observation is made that for real life applications "*it will be necessary to automate to the extent possible the process of discharging the proof obligations*." This is exactly what has been achieved to some extent in this chapter.

Chapter 9

Conclusions

In this thesis we have developed a verification theory and a tool for the automated analysis of assertions about object-oriented database schemas. The approach is inspired by the work of [SS89] in which a theorem prover is used to automate the verification of invariants for transactions on a relational database. The work presented in this thesis deals with an object-oriented database and it discusses applications other than the analysis of database transaction safety. An important difference with the work of [SS89] is that we have used a general purpose higher-order logic (HOL) theorem prover, namely the Isabelle theorem prover [Pau94, Isa], rather than implementing our own specialized prover. Much previous research, including the work of [SS89], concerns *fully* automatic techniques (i.e., without the possibility of further interaction). These techniques are inheritly limited in scope ([BGL96]). The presented approach, combines automatic and interactive proof, where Isabelle's automatic proof facilities are exploited to *minimize* the user's effort to discharge proof obligations. The results demonstrate that today's prover technology can indeed help in practical verification issues that arise in the design of databases.

This chapter concludes the thesis with a summary and a discussion of the achievements of this work. Finally, directions for further research are described.

9.1 Summary of results

9.1.1 A formal translation of object-oriented database schemas to HOL

The first goal of this research as stated in Chapter 1 was to develop a theory of object-oriented databases, which enables practical verification of various assertions about database schemas. This goal has been achieved in the first part of this thesis, where we define a *semantics* of object-oriented database schemas in HOL (see Chapter 5 and Appendix D). The semantics is the basis for the implementation of the schema translator described in Chapter 6. The schema translator implements a front-end tool for the Isabelle theorem prover. From a given database schema, the tool automatically generates a HOL theory with formal definitions of the class structures, methods, and the integrity constraints specified in the schema.

Concerning the representation of database schemas in HOL, an important issue has been the translation of the various object-oriented language features, including characteristic features such as inheritance, late binding, method overriding, and heterogeneous collections of objects. For the representation of these features, we have defined a simple theory of objects in HOL. This theory defines primitive operations on a store, independent of specific database structures. A number of theorems about these generic operations has been derived. These theorems play an important role in the practical verification work discussed in the second part of the thesis.

The data types of the database language other than objects could be mapped directly to similar constructs in HOL. For example, nested data structures (e.g., a list of sets of integer values) are already available in HOL. These data structures are characteristic of object-oriented databases, in contrast to the “flat” table structures found in relational databases.

The database language defined in Chapter 3 includes a relevant subset of concepts taken from object-oriented database programming. However, some types and operations have been excluded because they are not (yet) available in Isabelle. This includes the collection types bags and arrays (in addition to sets and lists, which are currently supported), and aggregates (e.g., sum, count, average — although Isabelle defines some of these operations for lists). The development of HOL theories that support these operations and types is actively addressed by the theorem-prover community. A preliminary theory of bags—multisets—is under development [Isa], but it is not yet sufficiently complete in the current Isabelle98-1 release. As new data type theories become available, they can be easily integrated, thanks to the extensible architecture of the Isabelle prover.

9.1.2 Semi-automated verification support using Isabelle

The second goal of this research was to apply the theoretical results developed in the first part of the thesis, to develop tool support for the verification of assertions about a database schema. Two applications have been studied.

In Chapter 7 we discussed a semi-automated technique to verify that methods defined in a database schema do not violate the specified global integrity constraints (database invariants). The analysis technique is based on the standard machinery of the Isabelle theorem prover and uses well-known verification techniques such as term-rewriting and deduction. The results have been published in [SB97, SE99].

In Chapter 8 we discussed other applications of the invariant analysis technique of Chapter 7 in the context of semantics-based transaction models, where other assertions about the semantics of database operations play a role. Particularly, we investigated the semi-automated verification of *compensation methods*. That is, for each method m, the designer of a system has to define another method m^{-1} to undo the effects of m. The analysis technique can be used to show that m followed by m^{-1} is the identity function. The results have been presented at the ECOOP’99 workshop on object-oriented databases [ES99].

The practical experiments discussed in this thesis indicate that, although inherently limited, automated verification is feasible and can be performed in a reasonable amount of time, for many non-trivial cases. The example database schema we have studied covers a diversity of object-oriented language features, with methods and constraints of various complexity.

It remains difficult, however, to characterize the kinds of method code for which the analysis can always be performed automatically (e.g., “methods with conditionals work” or “only updating the receiver object works”). A seemingly difficult proof may be found automatically, whereas a “trivial” one may fail.

The automated proof search performed by Isabelle is incomplete. Isabelle returns any subgoals that cannot be proved automatically. Typically, these goals should be examined further using Isabelle's interactive proof mode. However, detailed knowledge of both the theorem prover and the semantic embedding of the database language in HOL is needed at this stage. Some goals may be solved after a few more extra interactive steps, whereas others may turn out to be unprovable after all. Unprovable subgoals point to errors in the database schema, but identifying such errors from the remaining subgoals, and the subsequent correction of the code, remains a creative process that in the end requires human intelligence and skill. The analysis technique does not generate concrete counterexamples, although this may be a future possibility as theorem proving may eventually be combined with model checking (see [EMC96]). The generation of counterexamples is beyond the capabilities of the current generation of theorem provers.

9.2 Further work

In the following we list some directions for further research.

9.2.1 Generation of feedback

A desirable extension of this work is to improve on the generation of feedback. At present, the tool returns any subgoals that cannot be proved automatically. This kind of information is, however, of little help to users who do not have enough experience with the Isabelle system.

In [SMS87], several guidelines are given for the generation of feedback in the context of invariant analysis. Suggested feedback includes the generation of weakest and/or sufficient preconditions (also discussed briefly in Chapter 7). It amounts to assuming the unproved subgoals.

Due to the fact that reasoning takes place at the level of the semantics (i.e., at the level of the HOL representation, not at the conceptual level), an "inverse transformation" is needed from the semantics back to the conceptual notation. In essence, this is a problem in program transformation. Using Isabelle's pretty printing functionality, an attempt was made in this work to convert the HOL representation back to OASIS-like syntax (see Section 7.4.3), but at present only a limited number of operations is covered.

9.2.2 Generation of counterexamples

Theorem provers are often criticized of not being capable of generating counterexamples, in contrast to model checkers. Rather, they aim at proving program correctness. As pointed out in [EMC96], "*methods and tools should be optimized for finding errors, not certifying correctness. They should support generating counterexamples as a means of debugging.*"

An interesting direction for future work is to develop test-case generation techniques based on the subgoals that cannot be proved automatically by Isabelle. It involves choosing appropriate values for the quantified variables, based on the patterns in the goals. Isabelle's automated proof procedure has already narrowed down the search space: the remaining goals are the intricate cases where counterexamples may likely be found.

9.2.3 Commutativity analysis

Compensation, as discussed in Chapter 7 is typically combined with a notion of commutativity. The use of the present framework to support the verification of user-specified commutativity tables, including *backward commutativity* [Wei88, Wei93] and *forward commutativity* [Wei88, Wei93, LMWF94] must be further investigated. The initial experiments for backward commutativity as discussed briefly in Chapter 8 indicate that these proofs are more difficult than the proofs for invariant and compensation analysis. Worthwhile to investigate is the work on commutativity analysis by Rinard *et al* ([RD96a, RD96b, RD96c]).

Appendix A

Case Study: OASIS schema definition

In which we list the complete definition of the case study that is used as a running example throughout this thesis. The case study models part of the core functionality of the SEPIA cooperative authoring system [KAN94]. The OASIS schema definition contains definitions for editing a generic graph structure. Atomic nodes (class ANode) contain the hypermedia data of the document; composite nodes (class CNode) contain atomic nodes, links, and other composite nodes as elements. The schema serves to *structure* a hypermedia document. For this reason, the class hierarchy ends at the AtomicContents class, which is used for application-specific kinds and formats of media, such as text documents, audio, and graphics. Objects of the AtomicContents class maintain e.g., a file-system handle to the hypermedia object, along with system commands (represented as strings) for displaying and editing the object. Three persistent roots are specified, which serve as extents for classes CNodes, Link, and ANode. Furthermore, six integrity constraints are specified that should be preserved by the methods in the schema.

```
class AtomicContents {
  attribute string referenceDirectory;
  attribute string showStatement;
  attribute string URL;
  AtomicContents(string path, string stmt, string url) {
    referenceDirectory = path; showStatement = stmt; URL = url
  };
};

abstract class Element {
  attribute string name;
  attribute int position;
  abstract boolean isConnectedTo(Element n);
  string changeNameTo(string s) {
    string oldName = name; name = s;
    return oldName
  };
};
```

```
abstract class Node extends Element {
  attribute set⟨Link⟩ incomingLinks;
  attribute set⟨Link⟩ outgoingLinks;
  boolean addIncomingLink(Link k) {
    if k == nil or k in incomingLinks then { return false }
    else { incomingLinks += set(k) ; return true }
  };
  boolean addOutgoingLink(Link k) {
    if k == nil or k in outgoingLinks then { return false }
    else { outgoingLinks += set(k) ; return true }
  };
  void removeIncomingLink(Link k) { incomingLinks -= set(k) };
  void removeOutgoingLink(Link k) { outgoingLinks -= set(k) };
  boolean isConnectedTo(Link n) {
    return (n in incomingLinks) or (n in outgoingLinks)
  };
};

class ANode extends Node {
  attribute set⟨AtomicContent⟩ content;
  ANode(string s, int p) { name = s; position = p };
  AtomicContents createAtomicContentsIn(string path, string stmt, string url) {
    ac = new AtomicContents(path, stmt, url);
    content += set(ac);
    return (ac)
  };
  ANode removeAtomicContent(AtomicContent ac) {
   if (ac in content) then {
      content -= set(ac);
      return this
   } else return nil
  };
};

class CNode extends Node {
  attribute int size;
  attribute set⟨Element⟩ elements;
  CNode(string s, int p, int z) { name = s;  position = p; size = z };
  ANode createANodeIn(string s, int p) {
    ANode n = new ANode(s, p);
    elements += set(n);
    anodes += set(n);
    return n
  }
  boolean removeElement(Element n) {
    if (n != nil) and (n in elements) and
        (forall x in elements : not(x.isConnectedTo(n))) then {
          elements -= set(n); return true
    } else return false
  };
  Link createLinkIn(Node na, Node nb, string s, int p) {
    if (na in elements and nb in elements) then {
        Link n = new Link(s);
        n.from = n1; n.to = n2;
        elements += set(n);
        links += set(n);
        return n
    } else return nil
  };
  CNode createCNodeIn(int p, int z, string s) {
    CNode n = new CNode(s, p, z);
    elements += set(n);
    cnodes += set(n);
    return n
  };
};

class Link extends Element {
  attribute Node from;
  attribute Node to;
  Link(string s) { name = s; position = p; from = na; to = nb };
  boolean isConnectedTo(Node n) { return (from == n) or (to == n) };
};

name set⟨CNode⟩ cnodes;
name set⟨Link⟩ links;
name set⟨ANode⟩ anodes;

constraints {
 c1 : forall n in cnodes : n!=nil and (forall e in n.elements : e!=nil);
 c2 : forall n in links : n!=nil;
 c3 : forall n in anodes : n!=nil;
 c4 : forall n in cnodes : forall e in n.elements :
          ((e instanceof Link) implies
                (((Link)(e).from in n.elements) and
                   ((Link)(e).to in n.elements)));
 c5 : forall n1 in cnodes : forall n2 in cnodes :
          n1 == n2 or (forall n in n1.elements: not(n in n2.elements));
 c6 : forall n in anodes : forall e in n.incomingLinks : e!=nil;
};
```

Appendix B

OASIS Syntax definition

In which we define the syntax of the schema definition language OASIS-ODL, the object query language OASIS-OQL, and the object manipulation language OASIS-OML. Abstract syntax definitions are given using the following BNF notation:

- *Id* is the syntax of an identifier
- *x_literal* is a literal of type x, e.g. 125 is an int-literal.
- Non-terminals appear in italics font, e.g. *BoolExp*
- [*Symbol*] square brackets enclose an optional occurrence of a construct.
- Terminals of the grammar appear in sans serif font, e.g. constraint
- $\lceil$*Symbol*$\rceil$ top-brackets enclose a sequence of zero or more occurrences of a construct.

B.1 OASIS-ODL Schema

schema	::=	module *Id* $\{\lceil$definition$\rceil\}$	
definition	::=	class *Id* [extends *Id*] $\{[$*op*$\lceil;$ *op*$\rceil]\}$	(class definition)
	$\mid$	name T *Id*	(persistent root declaration)
	$\mid$	constraints $\{[$*expr*$\lceil;$ *expr*$\rceil]\}$	(constraint definitions)
	$\mid$	transactions $\{[$*umeth*$\lceil;$ *umeth*$\rceil]\}$	(transaction definitions)
op	::=	attribute T *Id*	(attribute declaration)
	$\mid$	*Id*$([$*pardecl*$,\ \lceil$*pardecl*$\rceil])$	(constructor definition)
	$\mid$	*cmeth* $\mid$ *ameth*	
cmeth	::=	OT *Id*$([$*pardecl*$,\ \lceil$*pardecl*$\rceil])\{$*com*$\}$	(method definition)
ameth	::=	OT *Id*$([$*pardecl*$,\ \lceil$*pardecl*$\rceil])$	(abstract method declaration)
pardecl	::=	T *Id*	(parameter declaration)

T	::=	int	
	$\mid$	bool	
	$\mid$	char	
	$\mid$	string	
	$\mid$	struct(*Id*:T_1 $\lceil$, *Id*:$T_2\rceil$)	
	$\mid$	set$\langle T\rangle$	
	$\mid$	list$\langle T\rangle$	
	$\mid$	*Id*	(types)

$$\begin{array}{rcll}
BT & ::= & \textsf{int} & \\
 & | & \textsf{bool} & \\
 & | & \textsf{char} & \\
 & | & \textsf{string} & \text{(basic types)}
\end{array}$$

$$\begin{array}{rcl}
OT & ::= & T \mid \textsf{void} \quad \text{(output types)}
\end{array}$$

B.2 OASIS-OML Commands

$$\begin{array}{rcll}
\textit{com} & ::= & L\ \textit{assignop exporcom} & \text{(variable and attribute update)} \\
 & | & \textsf{new}\ \textit{Id}([\textit{expr}\lceil, \textit{expr}\rceil]) & \text{(object creation)} \\
 & | & \textit{expr}.\textit{Id}([\textit{expr}\ \lceil, \textit{expr}\rceil]) & \text{(update method call)} \\
 & | & \textit{com}\ ;\ \textit{com} & \text{(sequential composition)} \\
 & | & T\ Id = \textit{exporcom}; \textit{com} & \text{(local variable declaration)} \\
 & | & \textsf{foreach}\ \textit{Id}\ \textsf{in}\ \textit{expr}\ [\textsf{where}\ \textit{expr}]\ \textsf{do}\ \textit{com} & \text{(collection iteration)} \\
 & | & \textsf{if}\ \textit{expr}\ \textsf{then}\ \textit{com}\ \textsf{else}\ \textit{com} & \text{(conditional)} \\
 & | & \textsf{return}\ \textit{expr} & \text{(return value)} \\
 & | & \textsf{skip} & \text{(no operation)}
\end{array}$$

$$\begin{array}{rcl}
L & ::= & \textit{expr}.\textit{Id} \mid \textit{Id} \\
\textit{assignop} & ::= & \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \\
\textit{exporcom} & ::= & \textit{expr} \mid \textit{com}
\end{array}$$

B.3 OASIS-OQL Query Expressions

$$\begin{array}{rcll}
\textit{expr} & ::= & \textit{LitExp} & \text{(Basic Literals (see Section 3.2.2))} \\
 & | & \textit{CLitExp} & \text{(Constructed Values (see Section 3.2.3))} \\
 & | & \textit{ArithExp} & \text{(Arithmetic Expressions (see Section 3.2.6)} \\
 & | & \textit{BoolExp} & \text{(Boolean Expressions (see Section 3.2.7))} \\
 & | & \textit{AccessExp} & \text{(Accessor Expressions (see Section 3.2.8))} \\
 & | & \textit{CollExp} & \text{(Collection Expressions (see Section 3.2.9))} \\
 & | & \textit{ListExp} & \text{(List\&String Expressions (see Section 3.2.10))} \\
 & | & \textit{SetExp} & \text{(Set Expressions (see Section 3.2.11))}
\end{array}$$

$$\begin{array}{rcl}
\textit{LitExp} & ::= & \textsf{nil} \mid \textsf{true} \mid \textsf{false} \\
 & | & \textit{int\_literal} \mid \textit{char\_literal} \mid \textit{string\_literal} \mid \textit{Id}
\end{array}$$

$$\begin{array}{rcl}
\textit{CLitExp} & ::= & \textsf{struct}(\textit{Id:expr}\ \lceil, \textit{Id:expr}\rceil) \\
 & | & \textsf{set}([\textit{expr}\ \lceil, \textit{expr}\rceil]) \mid \textsf{list}([\textit{expr}\ \lceil, \textit{expr}\rceil])
\end{array}$$

$$\begin{array}{rcl}
\textit{ArithExp} & ::= & \textit{expr}\ \texttt{+}\ \textit{expr} \mid \textit{expr}\ \texttt{-}\ \textit{expr} \\
 & | & \textit{expr}\ \texttt{*}\ \textit{expr} \mid \textit{expr}\ \texttt{/}\ \textit{expr} \\
 & | & \texttt{-}(\textit{expr})
\end{array}$$

$$\begin{array}{lcl}
\mathit{BoolExp} & ::= & \mathit{expr}\ \texttt{==}\ \mathit{expr} \mid \mathit{expr}\ \texttt{!=}\ \mathit{expr} \mid \\
 & \mid & \mathit{expr} < \mathit{expr} \mid \mathit{expr} > \mathit{expr} \\
 & \mid & \mathit{expr} <= \mathit{expr} \mid \mathit{expr} >= \mathit{expr} \\
 & \mid & \mathit{expr}\ \mathsf{and}\ \mathit{expr} \mid \mathit{expr}\ \mathsf{or}\ \mathit{expr} \mid \mathit{expr}\ \mathsf{implies}\ \mathit{expr} \mid \\
 & \mid & \mathsf{not}(\mathit{expr}) \\
 & \mid & \mathsf{forall}\ \mathit{Id}\ \mathsf{in}\ \mathit{expr} : \mathit{expr} \\
 & \mid & \mathsf{exists}\ \mathit{Id}\ \mathsf{in}\ \mathit{expr} : \mathit{expr}
\end{array}$$

$$\begin{array}{lcll}
\mathit{AccessExp} & ::= & \mathit{expr}\ .\ \mathit{Id} & \text{(attribute selection)} \\
 & \mid & \mathit{expr}\ .\ \mathit{Id}([\mathit{expr}\ \lceil, \mathit{expr}\rceil]) & \text{(method application)} \\
 & \mid & (\mathit{Id})\mathit{expr} & \text{(casting)} \\
 & \mid & \mathit{expr}\ \mathsf{instanceof}\ \mathit{Id} & \text{membership testing}
\end{array}$$

$$\begin{array}{lcl}
\mathit{CollExp} & ::= & \mathsf{select}\ \mathit{expr}\ \mathsf{from}\ \mathit{Id}\ \mathsf{in}\ \mathit{expr}\ \lceil, \mathit{Id}\ \mathsf{in}\ \mathit{expr}\rceil\ \mathsf{where}\ (\mathit{expr}) \\
 & \mid & \mathsf{element}(\mathit{expr}) \\
 & \mid & \mathsf{flatten}(\mathit{expr})
\end{array}$$

$$\begin{array}{lcl}
\mathit{ListExp} & ::= & \mathsf{head}(\mathit{expr}) \mid \mathsf{tail}(\mathit{expr}) \\
 & \mid & \mathsf{listtoset}(\mathit{expr})
\end{array}$$

$$\begin{array}{lcl}
\mathit{SetExp} & ::= & \mathit{expr}\ \mathsf{except}\ \mathit{expr} \\
 & \mid & \mathit{expr}\ \mathsf{union}\ \mathit{expr} \\
 & \mid & \mathit{expr}\ \mathsf{intersect}\ \mathit{expr}
\end{array}$$

Appendix C

OASIS Typing rules

In which typing rules are defined for the OASIS language. The typing rules assign a unique so-called *minimum type* (see [AC96]) to an expression or command. There is no subsumption rule [AC96]. Instead, typing rules include explicit clauses such that at any place where an expression of a supertype is expected, an expression of a subtype can be used instead. For example, the typing rule for method application requires that the receiver is a subtype of the class (type) in which the method is defined.

C.1 Preliminary definitions and notation

In the typing rules for expressions and commands, we use some auxiliary definitions and notation, which is explained below.

C.1.1 Schema context

The typing of expressions and commands is defined w.r.t. a given schema context. We assume the following operations on the schema:

- isa is the extends relation defined by the current schema (including the type Object, at the top of the hierarchy); $isa^{\star}$ is the closure of isa.

- $Cl = ACl \cup CCl$, where ACl and CCl represents the set of concrete and abstract classes in the current schema respectively.

- $lookup(C, op)$ is the operation (i.e. method or attribute) with name op that is most specific w.r.t. class C; i.e., $lookup(C, op)$ is the operation with name op defined in C if C defines op; otherwise, $lookup(C, op)$ is $lookup(C', op)$, where C' is the parent class of C.

C.1.2 Types of expressions and commands

For the typing of expressions and commands, we use a different grammar of types, than the one used for schema definitions (see Section B.1). A different grammar is needed because extra types are needed for typing the empty collections (type $*$ is used) and the nil literal (type

Nil is used). These types cannot be used in schema definitions (see Appendix B).

$$
\begin{array}{rcl}
\textit{typ} & ::= & BT \\
 & | & \mathsf{struct}(\textit{Id}:\textit{typ}\ \lceil, \textit{Id}:\textit{typ}\rceil) \\
 & | & \mathsf{set}\langle \textit{typ} \rangle \\
 & | & \mathsf{list}\langle \textit{typ} \rangle \\
 & | & Id \\
 & | & \mathsf{Nil} \\
 & | & *
\end{array}
$$

C.1.3 Environments

The typing rules are parameterized with an environment variable A of type env. Environments keep track of free variables that occur in an expression or a command. For environments, we use standard list notation: i.e., [] denotes the empty environment, and $(x, t) :: A$ is used for adding a new variable x of type t to an existing environment A.

C.1.4 Subtyping

The subtyping relation '$\leq$' is inductively defined on $typ \times typ$ as follows:

[sub$_1$] $$\frac{\tau \in BT}{\tau \leq \tau}$$

[sub$_2$] $$* \leq \tau$$

[sub$_3$] $$\frac{(C, C') \in \textbf{isa}^*}{C \leq C'}$$

[sub$_4$] $$\frac{C \in Cl}{\mathsf{Nil} \leq C}$$

[sub$_5$] $$\frac{\tau_1 \leq \tau_1' \cdots \tau_n \leq \tau_n'}{\mathsf{struct}(a_1{:}\tau_1, \cdots, a_n{:}\tau_n) \leq \mathsf{struct}(a_1{:}\tau_1', \cdots, a_n{:}\tau_n')}$$

[sub$_6$] $$\frac{\tau \leq \tau'}{\mathsf{set}\langle \tau \rangle \leq \mathsf{set}\langle \tau' \rangle}$$

[sub$_7$] $$\frac{\tau \leq \tau'}{\mathsf{list}\langle \tau \rangle \leq \mathsf{list}\langle \tau' \rangle}$$

These rules are used for typing the assignment operation.

C.1.5 Least-upper bound

The least-upper bound (LUB) of two types is defined as a partial function $\sqcup \in typ \times typ \hookrightarrow typ$, as follows:

[lub$_1$] $\tau \sqcup \tau = \tau$ for $\tau \in BT$

[lub$_2$] $* \sqcup \tau = \tau$

[lub$_3$] $\tau \sqcup * = \tau$

[lub$_4$] $C_1 \sqcup C_2 = C_3$, if $C_1, C_2 \in Cl$ and $C_3 = min\{C \in Cl \mid C_1 \leq C \wedge C_2 \leq C\}$

[lub$_5$] $\mathsf{Nil} \sqcup C = C$

[lub$_6$] $C \sqcup \mathsf{Nil} = C$

[lub$_7$] $\mathsf{set}\langle\tau\rangle \sqcup \mathsf{set}\langle\tau'\rangle = \mathsf{set}\langle\tau \sqcup \tau'\rangle$

[lub$_8$] $\mathsf{list}\langle\tau\rangle \sqcup \mathsf{list}\langle\tau'\rangle = \mathsf{list}\langle\tau \sqcup \tau'\rangle$

[lub$_9$] $\mathsf{struct}(a_1{:}\tau_1, \cdots, a_n{:}\tau_n) \sqcup \mathsf{struct}(a_1{:}\tau'_1, \cdots, a_n{:}\tau'_n) =$
$\mathsf{struct}(a_1{:}\tau_1 \sqcup \tau'_1, \cdots, a_n{:}\tau_n \sqcup \tau'_n)$

These rules are used for typing many binary operations of the OASIS-OQL expression language.

C.2 OASIS-OQL

Minimum typing of OASIS-OQL expressions is defined by the infix ':', which is a function of type $expr \rightarrow env \hookrightarrow typ$ defined inductively as follows.

[oql$_1$] $A \vdash$ *int_literal* : int

[oql$_2$] $A \vdash$ *char_literal* : char

[oql$_3$] $A \vdash$ *string_literal* : string

[oql$_4$] $A \vdash \mathsf{true} : \mathsf{bool}$

[oql$_5$] $A \vdash \mathsf{false} : \mathsf{bool}$

[oql$_6$] $A \vdash \mathsf{nil} : \mathsf{Nil}$

[oql$_7$] $x : t \vdash x : t$

[oql$_8$] $(x, t) :: A \vdash x : t$

[oql$_9$] $$\frac{A \vdash x : t}{(y, t') :: A \vdash x : t} \quad x \neq y$$

[oql$_{10}$] $$\frac{A \vdash e : t}{A \vdash \mathsf{struct}(a : e) : \mathsf{struct}(a : t)}$$

[oql$_{11}$] $$\frac{A \vdash e : t \qquad A \vdash \textsf{struct}(xs) : \textsf{struct}(ts)}{A \vdash \textsf{struct}(a : e, xs) : \textsf{struct}(a : t, ts)}$$

[oql$_{12}$] $A \vdash \textsf{set}() : \textsf{set}\langle * \rangle$

[oql$_{13}$] $$\frac{A \vdash e : t}{A \vdash \textsf{set}(e) : \textsf{set}\langle t \rangle}$$

[oql$_{14}$] $$\frac{A \vdash e : t_1 \qquad A \vdash \textsf{set}(es) : \textsf{set}\langle t_2 \rangle \qquad t_3 = t_1 \sqcup t_2}{A \vdash \textsf{set}(e, es) : \textsf{set}\langle t_3 \rangle}$$

[oql$_{15}$] $A \vdash \textsf{list}() : \textsf{list}\langle * \rangle$

[oql$_{16}$] $$\frac{A \vdash e : t}{A \vdash \textsf{list}(e) : \textsf{list}\langle t \rangle}$$

[oql$_{17}$] $$\frac{A \vdash e : t_1 \qquad A \vdash \textsf{list}(es) : \textsf{list}\langle t_2 \rangle \qquad t_3 = t_1 \sqcup t_2}{A \vdash \textsf{list}(e, es) : \textsf{list}\langle t_3 \rangle}$$

[oql$_{18}$] $$\frac{A \vdash e_1 : \textsf{int} \qquad A \vdash e_2 : \textsf{int}}{A \vdash e_1 \; op \; e_2 : \textsf{int}} \qquad op \in \{+, -, *, /\}$$

[oql$_{19}$] $$\frac{A \vdash e : \textsf{int}}{A \vdash \textsf{-}(e) : \textsf{int}}$$

[oql$_{20}$] $$\frac{A \vdash e_1 : t_1 \qquad A \vdash e_2 : t_2 \qquad t_1 \sqcup t_2 \text{ exists}}{A \vdash e_1 \; op \; e_2 : \textsf{bool}} \qquad op \in \{\textsf{==}, \textsf{!=}\}$$

[oql$_{21}$] $$\frac{A \vdash e_1 : t_1 \qquad A \vdash e_2 : \textsf{col}\langle t_2 \rangle \qquad t_1 \sqcup t_2 \text{ exists}}{A \vdash e_1 \; \textsf{in} \; e_2 : \textsf{bool}}$$

where $\textsf{col}\langle t \rangle$ denotes a collection type; i.e., set or list type

[oql$_{22}$] $$\frac{A \vdash e_1 : t_1 \qquad A \vdash e_2 : t_2 \qquad t = t_1 \sqcup t_2 \text{ exists} \qquad intorset(t)}{A \vdash e_1 \; op \; e_2 : \textsf{bool}} \qquad op \in \{<, >, <=, >=\}$$

where $intorset(t)$ is *true* if t is a set or an integer type, and *false* otherwise.

[oql$_{23}$] $$\frac{A \vdash e_1 : \textsf{bool} \qquad A \vdash e_2 : \textsf{bool}}{A \vdash e_1 \; op \; e_2 : \textsf{bool}} \qquad op \in \{\textsf{and}, \textsf{or}, \textsf{implies}\}$$

[oql$_{24}$] $$\frac{A \vdash e : \textsf{bool}}{A \vdash \textsf{not}(e) : \textsf{bool}}$$

[oql$_{25}$] $$\frac{A \vdash e : \textsf{col}\langle t \rangle \qquad (x, t) :: A \vdash p : \textsf{bool}}{A \vdash \textsf{forall} \; x \; \textsf{in} \; e{:}p : \textsf{bool}}$$

[oql$_{26}$] $$\frac{A \vdash e : \mathsf{col}\langle t\rangle \quad (x,t) :: A \vdash p : \mathsf{bool}}{A \vdash \mathsf{exists}\ x\ \mathsf{in}\ e{:}p : \mathsf{bool}}$$

[oql$_{27}$] $$\frac{A \vdash e : C}{A \vdash e.a : t} \quad \mathit{lookup}(C,a) \text{ is an attribute of type } t$$

[oql$_{28}$] $$\frac{A \vdash e : C \quad A \vdash e_i : t_i (i \in n) \quad t_i \leq t'_i (i \in n)}{A \vdash e \cdot m(e_1, \cdots, e_n) : t}$$

where $lookup(C,m)$ is a retrieval method with signature $t\ m(t'_1\ p_1, \cdots, t'_n\ p_n)$.

[oql$_{29}$] $$\frac{A \vdash e : C' \quad C \leq C'}{A \vdash (C)e : C}$$

[oql$_{30}$] $$\frac{A \vdash e_2 : \mathsf{col}\langle t\rangle \quad (x,t) :: A \vdash e_1 : t' \quad (x,t) :: A \vdash e_3 : \mathsf{bool}}{A \vdash \mathsf{select}\ e_1\ \mathsf{from}\ x\ \mathsf{in}\ e_2\ \mathsf{where}(e_3) : \mathsf{set}\langle t'\rangle}$$

[oql$_{31}$] $$\frac{A \vdash e_2 : \mathsf{col}\langle t\rangle \quad (x,t) :: A \vdash \mathsf{select}\ e_1\ \mathsf{from}\ xs\ \mathsf{where}(e_3) : \mathsf{set}\langle t'\rangle}{A \vdash \mathsf{select}\ e_1\ \mathsf{from}\ x\ \mathsf{in}\ e_2, xs\ \mathsf{where}(e_3) : \mathsf{set}\langle t'\rangle}$$

[oql$_{32}$] $$\frac{A \vdash e : \mathsf{col}\langle t\rangle}{A \vdash \mathsf{element}(e) : t}$$

[oql$_{33}$] $$\frac{A \vdash e : \mathsf{col}_1\langle \mathsf{col}_2\langle t\rangle\rangle}{A \vdash \mathsf{flatten}(e) : t'} \quad t' = \begin{cases} \mathsf{list}\langle t\rangle & \text{if both collections are lists} \\ \mathsf{set}\langle t\rangle & \text{otherwise} \end{cases}$$

[oql$_{34}$] $$\frac{A \vdash e : \mathsf{list}\langle t\rangle}{A \vdash \mathsf{head}(e) : t}$$

[oql$_{35}$] $$\frac{A \vdash e : \mathsf{list}\langle t\rangle}{A \vdash \mathsf{tail}(e) : \mathsf{list}\langle t\rangle}$$

[oql$_{36}$] $$\frac{A \vdash e : \mathsf{list}\langle t\rangle}{A \vdash \mathsf{listtoset}(e) : \mathsf{set}\langle t\rangle}$$

[oql$_{37}$] $$\frac{A \vdash e_1 : \mathsf{set}\langle t_1\rangle \quad A \vdash e_2 : \mathsf{set}\langle t_2\rangle \quad t = t_1 \sqcup t_2}{A \vdash e_1\ \mathsf{union}\ e_2 : \mathsf{set}\langle t\rangle}$$

[oql$_{38}$] $$\frac{A \vdash e_1 : \mathsf{set}\langle t_1\rangle \quad A \vdash e_2 : \mathsf{set}\langle t_2\rangle \quad t = t_1 \sqcup t_2}{A \vdash e_1\ \mathsf{intersect}\ e_2 : \mathsf{set}\langle t\rangle}$$

[oql$_{39}$] $$\frac{A \vdash e_1 : \mathsf{set}\langle t_1\rangle \quad A \vdash e_2 : \mathsf{set}\langle t_2\rangle \quad t = t_1 \sqcup t_2}{A \vdash e_1\ \mathsf{except}\ e_2 : \mathsf{set}\langle t\rangle}$$

In the above definition, we use recursive typing rules for operations that take a sequence of subexpressions. For example, the grammar rule for enumerated sets (set) defines three possible cases: the case that there are no elements, the case that there is just one element, and the case that there is more than one element. For each of these cases, a typing rule is given. The typing rules thus follow the structure of the BNF notation.

C.3 OASIS-OML

Minimum typing of OASIS-OML commands is defined by the infix ':' of type $com \to env \hookrightarrow typ \cup \{\textsf{void}\}$, inductively as follows

[oml$_1$] $$\frac{A \vdash l : t \quad A \vdash c : t' \quad t' \leq t}{A \vdash l = c : \textsf{void}}$$

[oml$_2$] $$\frac{A \vdash c_1 : t_1 \quad A \vdash c_2 : t_2}{A \vdash c_1 ; c_2 : t_2}$$

[oml$_3$] $$\frac{A \vdash c_1 : t_1 \quad (x_1, t_1) :: A \vdash c_2 : t_2}{A \vdash t_1\ x_1 = c_1 ; c_2 : t_2}$$

[oml$_4$] $$\frac{A \vdash e : \textsf{col}\langle t_1 \rangle \quad (x, t_1) :: A \vdash c : t_2}{A \vdash \textsf{foreach}\ x\ \textsf{in}\ e\ \textsf{do}\ c : t_3} \qquad t_3 = \begin{cases} \textsf{set}\langle t_2 \rangle & \text{if } t_2 \neq \textsf{void} \\ \textsf{void} & \text{otherwise} \end{cases}$$

[oml$_5$] $$\frac{A \vdash e : \textsf{col}\langle t_1 \rangle \quad A, x : t_1 \vdash p : \textsf{bool} \quad (x, t_1) :: A \vdash c : t_2}{A \vdash \textsf{foreach}\ x\ \textsf{in}\ e\ \textsf{where}\ p\ \textsf{do}\ c : t_3}$$

where t_3 is $\textsf{set}\langle t_2 \rangle$ if $t_2 \neq \textsf{void}$, and t_3 is $\textsf{void}$ otherwise.

[oml$_6$] $$\frac{A \vdash p : \textsf{bool} \quad A \vdash c_1 : t_1 \quad A \vdash c_2 : t_2 \quad t = t_1 \sqcup t_2}{A \vdash \textsf{if}\ p\ \textsf{then}\ c_1\ \textsf{else}\ c_2 : t}$$

[oml$_7$] $$\frac{C \in CCl}{A \vdash \textsf{new}\ C : C}$$

[oml$_8$] $$\frac{C \in CCl \quad A \vdash e_i : t_i (i \in n) \quad t_i \leq t'_i (i \in n)}{A \vdash \textsf{new}\ C(e_1, \cdots, e_n) : C}$$

where class C defines a constructor with signature $C(t'_1\ p_1, \cdots, t'_n\ p_n)$.

[oml$_9$] $$\frac{A \vdash e : C \quad A \vdash e_i : t_i (i \in n) \quad t_i \leq t'_i (i \in n)}{A \vdash e \cdot m(e_1, \cdots, e_n) : t}$$

where $lookup(C, m)$ is a method with signature $t\ m(t'_1\ p_1, \cdots, t'_n\ p_n)$.

[oml$_{10}$] $$\frac{A \vdash e : t}{A \vdash \textsf{return}\ e : t}$$

[oml$_{11}$] $A \vdash \textsf{skip} : \textsf{void}$

C.4 OASIS-ODL

The body of a method, constructor, transaction or constraints should be typed conform its signature declaration. For each class C, and for all methods $t\ m(t_1\ p_1, \cdots, t_n\ p_n)\{c\}$ defined in C, the following typing restriction applies. The initial environment is determined by the persistent roots $r_1 \cdots r_k$ with associated types $s_1 \cdots s_k$, the (inherited) attributes $a_1 \cdots a_\ell$ with associated types $u_1 \cdots u_\ell$ of C, and the method parameters. The method is type correct if and only if:

$$(p_n, t_n) :: \cdots :: (p_1, t_1) :: (a_\ell, u_\ell) :: \cdots :: (a_1, u_1) :: (r_k, s_k) :: \cdots :: (r_1, s_1) :: [\,] \vdash c : t$$

Note the sequence in which the variables are placed in the initial environment. Parameter names hide attribute names, which in turn hide root names. A similar rule is defined for class constructors, transactions, and constraints.

Appendix D

OASIS Semantics Rules

In which semantics rules are defined for types, expressions, and commands. The target of the translation is the pseudo Isabelle/HOL notation of Chapter 4, plus the constructs of the theory of objects discussed in Section 5.5. A detailed discussion of the rules and examples are found in Chapter 5.

D.1 Mapping of types

The HOL semantics of OASIS types $\mathcal{T} : typ \rightarrow holtyp$ is defined inductively as follows.

- $\mathcal{T}[\![\textsf{void}]\!] = \texttt{unit}$
- $\mathcal{T}[\![\textsf{bool}]\!] = \texttt{bool}$
- $\mathcal{T}[\![\textsf{int}]\!] = \texttt{int}$
- $\mathcal{T}[\![\textsf{string}]\!] = \texttt{string}$
- $\mathcal{T}[\![\textsf{C}]\!] = \texttt{oid}$, for any class name $C \in Cl$
- $\mathcal{T}[\![\textsf{struct}(a_1 : t_1, \cdots, a_n : t_n)]\!] = \mathcal{T}[\![t_1]\!] \times \cdots \times \mathcal{T}[\![t_n]\!]$
- $\mathcal{T}[\![\textsf{set}\langle t\rangle]\!] = (\mathcal{T}[\![t]\!])\ \texttt{set}$
- $\mathcal{T}[\![\textsf{list}\langle t\rangle]\!] = (\mathcal{T}[\![t]\!])\ \texttt{list}$

D.2 Mapping of OASIS-OQL expressions

The HOL semantics of OASIS-QOL expressions $\mathcal{E} \in expr \rightarrow env \rightarrow holexp$ is inductively defined as follows.

- $\mathcal{E}[\![\textsf{true}]\!]_A = \texttt{True}$
- $\mathcal{E}[\![\textsf{false}]\!]_A = \texttt{False}$
- $\mathcal{E}[\![\textsf{nil}]\!]_A = \texttt{None}$
  Our preprocessor, however, takes e != nil as e instanceof C (for C the type of e), which leads to better HOL code; a similar reduction applies to e == nil.
- $\mathcal{E}[\![i]\!]_A = i$, for i an integer constant

- $\mathcal{E}[\![x]\!]_A = x$, for x a variable name
- $\mathcal{E}[\![\textsf{struct}(a_1 : e_1, \cdots, a_n : e_n)]\!]_A = (\mathcal{E}[\![e_1]\!]_A, \cdots, \mathcal{E}[\![e_n]\!]_A)$
- $\mathcal{E}[\![e \cdot a_i]\!]_A = \texttt{let}\ (a_1, \cdots, a_n) = (\mathcal{E}[\![e]\!]_A)\ \texttt{in}\ a_i)$
  whenever e is a record expression—not an object—of type $\textsf{struct}(a_1 : t_1, \cdots, a_n : t_n)$.
- $\mathcal{E}[\![\textsf{set}(e_1, \cdots, e_n)]\!]_A = \{\mathcal{E}[\![e_1]\!]_A, \cdots, \mathcal{E}[\![e_n]\!]_A\}$
- $\mathcal{E}[\![e_1\ \textsf{union}\ e_2]\!]_A = (\mathcal{E}[\![e_1]\!]_A) \cup (\mathcal{E}[\![e_2]\!]_A)$
- $\mathcal{E}[\![e_1\ \textsf{intersect}\ e_2]\!]_A = (\mathcal{E}[\![e_1]\!]_A) \cap (\mathcal{E}[\![e_2]\!]_A)$
- $\mathcal{E}[\![e_1\ \textsf{except}\ e_2]\!]_A = (\mathcal{E}[\![e_1]\!]_A) - (\mathcal{E}[\![e_2]\!]_A)$
- $\mathcal{E}[\![\textsf{select distinct}\ e\ \textsf{from}\ x_1\ \textsf{in}\ e_1, \cdots, x_k\ \textsf{in}\ e_k\ \textsf{where}\ e']\!]_A =$
  $$\{\mathcal{E}[\![e]\!]_A \mid (x_1, \ldots, x_k).\ x_1 \in (\mathcal{E}[\![e_1]\!]_{A_1}) \wedge \cdots \wedge x_k \in (\mathcal{E}[\![e_k]\!]_{A_k}) \wedge \mathcal{E}[\![e']\!]_A\}$$
  where $A_1 \equiv A$, and $A_i \equiv (x_{i-1}, t_{i-1}) :: A_{i-1}$ for $1 < i \leq k$.
- $\mathcal{E}[\![\textsf{listtoset}\ (e)]\!]_A = \texttt{list\_to\_set}(\mathcal{E}[\![e]\!]_A)$
- $\mathcal{E}[\![\textsf{flatten}\ (e)]\!]_A = \bigcup(\mathcal{E}[\![e]\!]_A)$
- $\mathcal{E}[\![\textsf{element}\ (e)]\!]_A = \epsilon z.z \in (\mathcal{E}[\![e]\!]_A)$
- $\mathcal{E}[\![e_1 = e_2]\!]_A = (\mathcal{E}[\![e_1]\!]_A) = (\mathcal{E}[\![e_2]\!]_A)$
- $\mathcal{E}[\![e_1\ != e_2]\!]_A = (\mathcal{E}[\![e_1]\!]_A) \neq (\mathcal{E}[\![e_2]\!]_A)$
- $\mathcal{E}[\![e_1 \leq e_2]\!]_A = (\mathcal{E}[\![e_1]\!]_A) \leq (\mathcal{E}[\![e_2]\!]_A)$
- $\mathcal{E}[\![e_1\ \textsf{in}\ e_2]\!]_A = (\mathcal{E}[\![e_1]\!]_A) \in (\mathcal{E}[\![e_2]\!]_A)$
- $\mathcal{E}[\![\textsf{forall}\ x\ \textsf{in}\ e_1 : e_2]\!]_A = \forall x \in (\mathcal{E}[\![e_1]\!]_A) \mid (\mathcal{E}[\![e_2]\!]_{(x,t)::A})$
  whenever $A \vdash e_1 : \texttt{col}\langle t \rangle$.
- $\mathcal{E}[\![\textsf{exists}\ x\ \textsf{in}\ e_1 : e_2]\!]_A = \exists x \in (\mathcal{E}[\![e_1]\!]_A) \mid (\mathcal{E}[\![e_2]\!]_{(x,t)::A})$ whenever $A \vdash e_1 : \texttt{col}\langle t \rangle$.
- $\mathcal{E}[\![\textsf{not}\ e]\!]_A = \neg(\mathcal{E}[\![e]\!]_A)$
- $\mathcal{E}[\![e\ \textsf{instanceof}\ C]\!]_A =$
  $\texttt{eval}\ (\texttt{os}\ \mathcal{E}[\![e]\!]_A)$
  $(\lambda\texttt{val.case val of}\ C_1\ a_{11} \cdots a_{1k_1} \Rightarrow d_1 \mid \cdots \mid C_n\ a_{n1} \cdots a_{nk_n} \Rightarrow d_n)$
  where $d_i = \texttt{True}$ if $C_i \leq C$, and $d_i = \texttt{False}$ otherwise.
- $\mathcal{E}[\![C(e)]\!]_A =\ \texttt{get}\ (\texttt{os}\ \mathcal{E}[\![e]\!]_A)$
  $(\lambda\texttt{val.case val of}\ C_1\ a_{11} \cdots a_{1k_1} \Rightarrow d_1 \mid \cdots \mid C_n\ a_{n1} \cdots a_{nk_n} \Rightarrow d_n)$
  where $d_i = (\mathcal{E}[\![e]\!]_A)$ if $C_i \leq C$, and $d_i = \texttt{arbitrary}$ otherwise.
- $\mathcal{E}[\![e \cdot a_k]\!]_A =\ \texttt{get}\ (\texttt{os}\ \mathcal{E}[\![e]\!]_A)$
  $(\lambda\texttt{val.case val of}\ C_1\ a_{11} \cdots a_{1k_1} \Rightarrow d_1 \mid \cdots \mid C_n\ a_{n1} \cdots a_{nk_n} \Rightarrow d_n)$
  where $d_i = a_{ik_i}$, for k_i the position of attribute a_k in class $C_i \leq C$, and $d_i = \texttt{arbitrary}$ otherwise.

- $\mathcal{E}[\![e \cdot m(e_1, \cdots, e_k)]\!]_A = ($ $\texttt{if}$ $\mathcal{E}[\![e \text{ instanceof } C_z]\!]_A$
  $\quad\texttt{then}\ C_z\_m\ \texttt{os}\ x_1 \cdots x_i\ \mathcal{E}[\![e]\!]_A\ \ \mathcal{E}[\![e_1]\!]_A \cdots \mathcal{E}[\![e_k]\!]_A$
  $\quad\texttt{else} \cdots (\ \texttt{if}\ \mathcal{E}[\![e \text{ instanceof } C_0]\!]_A$
  $\qquad\texttt{then}\ C_0\_m\ \texttt{os}\ x_1 \cdots x_i\ \mathcal{E}[\![e]\!]_A\ \ \mathcal{E}[\![e_1]\!]_A \cdots \mathcal{E}[\![e_k]\!]_A$
  $\qquad\texttt{else arbitrary}))$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, $e : C$, $i \leq n$ the number of persistent root names, C_0 is the "least"(w.r.t. the subtyping relation $\leq$) superclass of C that defines m, and $C_1 \cdots C_z$ the ordered sequence of subclasses of C that (re-)define m; the ordering is such that $C_v \leq C_w$ implies C_v is placed after C_w, for all $v \leq z$ and $w \leq z$.

D.3 Mapping of OASIS-OML commands

The HOL semantics of OASIS-OML commands $\mathcal{C} \in com \to env \to holexp$ is inductively defined as follows.

- $\mathcal{C}[\![\text{skip}]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!] \cdot$
  $(\texttt{skip}, \texttt{None} : (t_1\ \texttt{option}), \cdots, \texttt{None} : (t_n\ \texttt{option}), ())$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$

- $\mathcal{C}[\![\text{return } e]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!] \cdot$
  $(\texttt{skip}, \texttt{None} : (t_1\ \texttt{option}), \cdots, \texttt{None} : (t_n\ \texttt{option}), \mathcal{E}[\![e]\!]_A)$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$

- $\mathcal{C}[\![e \cdot a_\ell \oplus e_\ell]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!] \cdot$
  $(\texttt{set os}\ (\mathcal{E}[\![e]\!])$
  $\quad(\lambda\texttt{val} \cdot \texttt{case val of}\ C_1\ a_{11} \cdots a_{1k_1} \Rightarrow C_1\ d_{11} \cdots d_{1k_1}$
  $\qquad| \cdots$
  $\qquad| C_n\ a_{n1} \cdots a_{nk_n} \Rightarrow C_n\ d_{n1} \cdots d_{nk_n}),$
  $\ \texttt{None} : (t_1\ \texttt{option}), \cdots, \texttt{None} : (t_n\ \texttt{option}), ())$
  whenever $A = [(x_1, t_1), \cdots, (x_n, t_n)]$, $A \vdash e : C$, $d_{ij} = \begin{cases} op(\mathcal{E}[\![e_\ell]\!]_A, a_{ij}) & \text{if } j = \ell \wedge C_i \leq C \\ a_{ij} & \text{otherwise} \end{cases}$
  For $\oplus \equiv$ +=, the function *op* is defined as $\lambda(x, y) \cdot x \cup y$ if e_ℓ is a set; as $\lambda(x, y) \cdot x@y$ if e_ℓ is a list; and as $\lambda(x, y) \cdot x + y$ if e_ℓ is an integer. For $\oplus \equiv$ -=, the function *op* is defined as $\lambda(x, y) \cdot x - y$ and for $\oplus \equiv$ =, the function *op* is defined as $\lambda(x, y) \cdot x$.

- $\mathcal{C}[\![e \cdot a_\ell \oplus c]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_m \cdot$
  $\texttt{let}\ (\Delta, y_1, \cdots, y_n, \texttt{result}) = \mathcal{C}[\![c]\!]_A\ \texttt{os}\ x_1 \cdots x_n\ \mu_1 \cdots \mu_m$
  $\texttt{in}\ (\ \texttt{smash}\ \Delta\ (\texttt{set os}\ (\mathcal{E}[\![e]\!])$
  $\quad(\lambda\texttt{val} \cdot \texttt{case val of}\ C_1\ a_{11} \cdots a_{1k_1} \Rightarrow C_1\ d_{11} \cdots d_{1k_1}$
  $\qquad| \cdots$
  $\qquad| C_n\ a_{n1} \cdots a_{nk_n} \Rightarrow C_n\ d_{n1} \cdots d_{nk_n})), y_1, \cdots, y_n, ())$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, $A \vdash e : C$, m the number of oid generator parameters needed for c, and $d_{ij} = \begin{cases} op(\texttt{result}, a_{ij}) & \text{if } j = \ell \wedge C_i \leq C \\ a_{ij} & \text{otherwise} \end{cases}$
  The function *op* is defined as in the previous rule.

- $\mathcal{C}[\![\text{new } C(e_1, \cdots, e_\ell)]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_m\lambda\mu_{m+1} : \texttt{oid} \cdot$

$\texttt{let}\ (\Delta, y_1, \cdots, y_n, y_{n+1}, \texttt{result}) = C\_const\ (\texttt{smash os}\ (\texttt{new}\ \mu_{m+1}\ (C\ \ d_1 \cdots d_k)))$
$\quad x_1 \cdots x_i\ \mu_{m+1}\ [\![e_1]\!] \cdots \mathcal{E}[\![e_\ell]\!]\ \mu_1 \cdots \mu_m$
$\texttt{in}\ (\Delta, y_1, \cdots, y_i, \texttt{None} : \mathcal{T}[\![t_{i+1}]\!], \cdots, \texttt{None} : \mathcal{T}[\![t_n]\!], \mu_{m+1})$

whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, i the number of persistent root variables,and m the number of oid generator parameters needed for c.

- $\mathcal{C}[\![x_i \oplus e_i]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!] \cdot$
  $(\Delta, \texttt{None} : (t_1\ \texttt{option}), \cdots, \texttt{None} : (t_{i-1}\ \texttt{option}), d, \texttt{None} : (t_{i+1}\ \texttt{option}), \cdots, y_n, ())$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, $d = op(\mathcal{E}[\![e_i]\!]_A, x_i)$, and the function op is defined as in the previous rule.

- $\mathcal{C}[\![x_i \oplus c]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_m \cdot$
  $\texttt{let}\ (\Delta, y_1, \cdots, y_n, \texttt{result}) = \mathcal{C}[\![c]\!]_A\ \texttt{os}\ x_1 \cdots x_n\ \mu_1 \cdots \mu_m$
  $\texttt{in}\ (\Delta, y_1, \cdots, y_{i-1}, d, y_{i+1}, \cdots, y_n, ())$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, and m the number of oid generator parameters needed for c; $d = op(\texttt{result}, x_i)$, and the function op is defined as in the previous rule.

- $\mathcal{C}[\![c_1\ ; c_2]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_m\lambda\mu_{m+1} \cdots \lambda\mu_{m+k} \cdot$
  $\texttt{let}\ (\Delta_1, y_1, \cdots, y_n, r_1) = \mathcal{C}[\![c_1]\!]_A\ \texttt{os}\ x_1 \cdots x_n\ \mu_1 \cdots \mu_m;$
  $\quad (\Delta_2, z_1, \cdots, z_n, r_2) = \mathcal{C}[\![c_2]\!]_A\ (\texttt{smash os}\ \Delta_1)\ (\texttt{case}\ y_1\ \texttt{of None} \Rightarrow x_1 \mid \texttt{Some}\ y \Rightarrow y)$
  $\quad\quad \vdots$
  $\quad\quad (\texttt{case}\ y_n\ \texttt{of None} \Rightarrow x_n \mid \texttt{Some}\ y \Rightarrow y)$
  $\quad\quad \mu_{m+1} \cdots \mu_{m+k}$
  $\texttt{in}\ (\texttt{smash}\ \Delta_1\ \Delta_2, \texttt{case}\ z_1\texttt{of None} \Rightarrow y_1 \mid \texttt{Some}\ y \Rightarrow \texttt{Some}\ y, \cdots,$
  $\quad \texttt{case}\ z_n\texttt{of None} \Rightarrow y_n \mid \texttt{Some}\ y \Rightarrow \texttt{Some}\ y, ())$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, m the number of oid generator parameters needed for c_1, and k the number of oid generator parameters needed for c_2.

- $\mathcal{C}[\![t_{n+1}\ x_{n+1} = c_1; c_2]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_m \cdot$
  $\texttt{let}\ (\Delta, y_1, \cdots, y_n, y_{n+1}, \texttt{result}) =$
  $\quad \mathcal{C}[\![x_{n+1} = c_1; c_2]\!]_{(x_{n+1}, t_{n+1})::A}\ \texttt{os}\ x_1 \cdots x_n\ \texttt{arbitrary}\ \mu_1 \cdots \mu_m$
  $\texttt{in}\ (\Delta, y_1, \cdots, y_n, \texttt{result})$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, and m the number of oid generator parameters needed for $x_{n+1} = c_1; c_2$.

- $\mathcal{C}[\![\text{foreach}\ x\ \text{in}\ e\ \text{do}\ c]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_m \cdot$
  $\texttt{let}\ R = \mathcal{E}[\![e]\!]; \Delta_f = (\lambda x \cdot \mathcal{C}[\![c]\!]_{[\,]})$
  $\texttt{in}\ (\ \texttt{apply}\ R\ (\lambda x \cdot \texttt{fst}(\Delta_f\ \texttt{os}\ x\ (\mu_1\ x) \cdots (\mu_m\ x)),$
  $\quad \texttt{None} : (t_1\ \texttt{option}), \cdots, \texttt{None} : (t_n\ \texttt{option}), \{\texttt{snd}(\Delta_f\ \texttt{os x}) \mid \texttt{x} \cdot x \in R\})$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, m the number of oid generator parameters needed for c. Notice that variables cannot be updated within the loop, so that c takes the empty environment [].

- $\mathcal{C}[\![e \cdot m(e_1, \cdots, e_k)]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_{p_z + \cdots + p_0}$
  $\texttt{if}\ \mathcal{E}[\![e\ \text{instanceof}\ C_z]\!]_A$
  $\texttt{then}\ C_z\_m\ \texttt{os}\ x_1 \cdots x_i\ \mathcal{E}[\![e]\!]_A\ \ \mathcal{E}[\![e_1]\!]_A \cdots \mathcal{E}[\![e_k]\!]_A\mu_{1+p_{n-1}+\cdots p_0} \cdots \mu_{p_z + \cdots p_0}$
  $\texttt{else} \cdots (\ \texttt{if}\ \mathcal{E}[\![e\ \text{instanceof}\ C_0]\!]_A$
  $\quad \texttt{then}\ C_0\_m\ \texttt{os}\ x_1 \cdots x_i\ \mathcal{E}[\![e]\!]_A\ \ \mathcal{E}[\![e_1]\!]_A \cdots \mathcal{E}[\![e_k]\!]_A\mu_1 \cdots \mu_{p_0}$
  $\quad \texttt{else arbitrary})$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, $e : C$, $i \leq n$ the number of persistent roots, p_j the number of oid generator parameters needed for the C_j implementation of m, C_0 is the "least"(w.r.t. the sub-

typing relation $\leq$) superclass of C that defines m, and $C_1 \cdots C_z$ the ordered sequence of subclasses of C that (re-)defines m; the ordering is such that $C_v \leq C_w$ implies C_v is placed after C_w, for all $v \leq z$ and $w \leq z$.

- $\mathcal{C}[\![\textsf{if}\ (b)\ c_1\ \textsf{else}\ c_2\ ]\!]_A = \lambda\texttt{os} : \texttt{OS}\lambda x_1 : \mathcal{T}[\![t_1]\!] \cdots \lambda x_n : \mathcal{T}[\![t_n]\!]\lambda\mu_1 \cdots \lambda\mu_m\lambda\mu_{m+1} \cdots \lambda\mu_{m+k}$
  $\texttt{if}\ (\mathcal{E}[\![b]\!])\ \texttt{then}\ (\mathcal{C}[\![c_1]\!]_A\ \texttt{os}\ x_1 \cdots x_n\mu_1 \cdots \mu_m)\ \texttt{else}\ (\mathcal{C}[\![c_2]\!]_A\ \texttt{os}\ x_1 \cdots x_n\ \mu_m \cdots \mu_{m+k})$
  whenever $A = [(x_n, t_n), \cdots, (x_1, t_1)]$, m the number of oid generator parameters needed for c_1, and k the number of oid generator parameters needed for c_2.

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[AG96] Ken Arnold and James Gosling. *The Java programming Language*. Addison-Wesley, 1996.

[AJR97] Paul Ammann, Sushil Jajodia, and Indrakshi Ray. Applying formal methods to semantic-based decomposition of transactions. *ACM Transactions on Database Systems*, 22(2):215–254, June 1997.

[Bar84] H.P. Barendrecht. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[BD95] Véronique Benzaken and Anne Doucet. Thémis: A database programming language handling integrity constraints. *VLDB Journal*, 4(3):493–518, 1995.

[BdBZ93] H. Balsters, R. A. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In Oscar M. Nierstrasz, editor, *Proceedings of the Seventh European Conference on Object-Oriented Programming*, volume 707 of *LNCS*, pages 161–184, Kaiserslautern, Germany, 1993. Springer.

[BDK92] Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-oriented Database System: The Story of O2*. Morgan Kaufmann, 1992.

[BDM93] F. Bry, H. Decker, and R. Manthey. A uniform approach to constrain satisfaction and constraint satisfiability in deductive databases. In *Proceedings of the 4th Australian Database Conference*, pages 161–170, Brisbane, Australia, 1993.

[BF91] Herman Balsters and Maarten M. Fokkinga. Subtyping can have a simple semantics. *Theoretical Computer Science*, 87:81–96, September 1991.

[BGG$^+$92] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*. North-Holland, June 1992.

[BGL96] M. Benedikt, T. Griffin, and L. Libkin. Verifiable properties of database transactions. In *Proceedings of Principles of Database Systems (PODS)*, pages 117–127, 1996.

[BS96] V. Benzaken and X. Schaefer. Ensuring efficiently the integrity of a persistent object store via abstract interpretation. In *Workshop on Persistent Object Systems (POS)*, pages 72–87, 1996.

[BS97] V. Benzaken and X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In *Proceedings of ECOOP*, volume 1241 of *LNCS*, pages 60–85. Springer-Verlag, 1997.

[BS98] V. Benzaken and X. Schaefer. Static management of integrity in object-oriented databases: Design and implementation. In *Extending Database Technology (EDBT)*, March 1998.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and computing*, 76:138–164, 1988.

[CB97] R. G. G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[CD97] Q. Chen and U. Dayal. Failure handling for transaction hierarchies. In *Proceedings of ICDE*, pages 245–254, Birmingham, U.K., April 1997.

[CM94] Luca Cardelli and John Mitchell. Operations on records. In C. Gunther and J.C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programmin*. MIT press, 1994.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[DH95] M. Doherty and R. Hull. Towards a framework for efficient management of potentially conflicting database updates. In *IFIP International Conference on Database Semantics (DS-6)*, Atlanta, Georgia, May 1995.

[DHDD95] M. Doherty, R. Hull, M. Derr, and J. Durand. On detecting conflict between proposed updates. In *International Workshop on Database Programming Languages (DBPL)*, Gubbio, Italy, September 1995.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[EFK$^+$96] S. Even, F. Faase, H. Kaijanranta, J. Klingemann, A. Lehtola, O. Pihlajamaa, T. Tesch, and J. Wäsch. Deliverable VII.1: Design of the TransCoop Demonstrator System. Report TC/REP/GMD/D7-1/704, Esprit Project No. 8012, 1996.

[EGLT76] K. Eswaran, Jim Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database systems. *Communications of the ACM*, 19(11):624–633, Nov 1976.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

[EMC96] Jeannette Wing Edmund M. Clarke. Formal methods: state of the art and future directions. Technical report, Carnegie Mellon University, August 1996.

[ES99] Susan Even and David Spelt. Compensation methods to support generic graph editing: A case study in automated verification of schema requirements for an advanced transaction model. In *Proceedings of the 1st ECOOP workshop on object-oriented databases*, Lisbon, Portugal, 15 June 1999.

[FP93] Piero Fraternali and Stefano Paraboschi. A review of repairing techniques for integrity maintenance. In *Rules in database systems*, pages 333–346. Springer, 1993.

[GM79] Gardarin and Melkanoff. Proving consistency of database transactions. In *Proceedings of the 5th VLDB Conference*, pages 291–298, Rio de Janeiro, Brazil, 1979.

[HI85] Arding Hsu and Tomasz Imielinski. Integrity checking of multiple updates. *ACM SIGMOD*, pages 152–168, 1985.

[Isa] Isabelle. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html`.

[JK99] Bill Joy and Ken Kennedy. Information technology research: Investing in our future. In *President's Information Technology Advisory Committee (PITAC) Report to the President*. `http://www.ccic.gov/ac/report`, 1999.

[JQ92] H. V. Jagadish and Xiaolei Qian. Integrity maintenance in an object-oriented database. In *Proceedings of the 18th VLDB Conference*, pages 469–480, Vancouver, British Columbia, 1992.

[JvdBH$^+$98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java Classes (Preliminary Report). In *Proceedings of OOPSLA*, 1998. To appear.

[KAN94] W. Klas, K. Aberer, and E. J. Neuhold. Object-Oriented Modelling for Hypermedia Systems Using the VODAK Modelling Language (VML). In *Advances in Object-Oriented Database Systems*, volume 130 of *Computer and Systems Sciences*, pages 389–443. Springer-Verlag, 1994.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th VLDB Conference*, pages 95–106, Brisbane, Australia, 1990.

[KTWK97] Justus Klingemann, Thomas Tesch, Jürgen Wäsch, and Wolfgang Klas. The TransCoop Transaction Model. In *Transaction Management Support for Cooperative Applications*, chapter 7, pages 149–172. Kluwer Academic Publishers, 1997.

[Law95] M. Lawley. Transaction safety in deductive object-oriented databases. In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*, volume 1013 of *LNCS*, pages 395–410, Singapore, 1995.

[LMWF94] Nancy Lynch, Michael Merrit, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

[LS93] C. Laasch and M. H. Scholl. Deterministic semantics of set-oriented update sequences. In *Proceedings of ICDE*, pages 4–13, Vienna, Austria, April 1993.

[LT93] Michael Lawley and Rodney Topor. Using weakest preconditions to simplify integrity constraint checking. In *Proceedings of the 4th Australian Database Conference*, pages 161–170, Brisbane, Australia, 1993.

[Mey85] Bertrand Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1985.

[Mey97] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1997.

[MR97] Cris Pedregal Martin and Krithi Ramamritham. Delegation: Efficiently rewriting history. In *Proceedings of ICDE*, pages 266–275, Birmingham, U.K., April 1997.

[MS76] R. Milner and C. Strachey. *A Theory of Programming Language Semantics*. Wiley, New York, 1976.

[Nic82] Jean-Marie Nicolas. Logic for improving integrity checking in relational data base. *Acta Informatica*, 18:227–253, 1982.

[Nip98] Topias Nipkow. Tutorial on Isabelle/HOL, 1998. Electronic document available from `http://www.cl.cam.ac.uk/users/lcp/ml-aftp/doc/tutorial.pdf`.

[NW98] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcolm Newey, editors, *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '98)*, LNCS, Canberra, Australia, August 1998. Springer-Verlag.

[ORR$^+$96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. Pvs: Combining specification, proof checking, and model checking. In *Computer-Aided Verification (CAV '96)*, number 1102 in LNCS, July/August 1996.

[Özsu94] M. Tamer Özsu. Transaction models and transaction management in object-oriented database management systems. In *Advances in Object-Oriented Database Systems*, volume 130 of *Computer and Systems Sciences*, pages 147–184. Springer-Verlag, 1994.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Pau90] Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau93a] Lawrence C. Paulson. The Isabelle Reference Manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[Pau93b] Lawrence C. Paulson. Isabelle's Object-Logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer (2nd ed.)*. Cambridge University Press, New York, NY, 1996.

[Qia88] Xiaolei Qian. An effective method for integrity constraint simplification. In *Fourth International Conference on Data Engineering*, pages 338–345, Los Angeles, California, 1988.

[Qia91] Xiaolei Qian. The expressive power of the bounded-iteration construct. *Acta Informatica*, 28(7):631–656, October 1991.

[Qia93] Xiaolei Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems*, 18(4):626–677, December 1993.

[RD96a] M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the SIGPLAN Conference on Program Language Design and Implementation*, 1996.

[RD96b] M. Rinard and P. Diniz. Commutativity analysis: A technique for automatically parallelizing pointer-based computations. In *Proceedings of 10th International Parallel Processing Symposium*, 1996.

[RD96c] M. Rinard and P. Diniz. Semantic foundations of commutativity analysis. In *Second International Euro-Par Conference*, Lyon, France, August 1996.

[RKT$^+$95] Marek Rusinkiewicz, Wolfgang Klas, Thomas Tesch, Jürgen Wäsch, and Peter Muth. Towards a cooperative transaction model—The Cooperative Activity Model. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, September 1995.

[San97] Thomas Santen. A theory of structured model-based specifications in Isabelle/HOL. In *Proc. of the 1997 International Conference on Theorem Proving in Higher Order Logics (TPHOLs97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[SB97] David Spelt and Herman Balsters. Automatic verification of transactions on object-oriented databases. In *Proceedings of the Workshop on Database Programming Languages (DBPL)*, Estes Park, Colorado, 1997.

[SE99] David Spelt and Susan Even. A theorem prover-based analysis tool for object-oriented databases. In *Tools and Algorithms for the Construction and Analysis of Systems, 5th International Conference, TACAS'99*, number 1579 in LNCS, pages 375–389. Springer-Verlag, 1999.

[Sel95] Ron Seljée. A new method for integrity constraint checking in deductive database. *Data & Knowledge Engineering*, 15(1):63–102, March 1995.

[SMS87] David Stemple, Subhasish Mazumdar, and Tim Sheard. On the modes and menaing of feedback to transaction designers. *ACM SIGMOD Record*, 16(3):374–386, December 1987.

[Spe95] David Spelt. A Proof Tool for TM. M.Sc. Thesis, Universiteit Twente, July 1995.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual, Second edition*. Prentice Hall International, 1992.

[SQL92] SQL2. Information technology—database languages—SQL. Technical Report ISO/IEC 9075, ISO, 1992. `http://www.jcc.com/sql_stnd.html`.

[SS89] Tim Sheard and David Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[Too97] Werner L. Toonk. A mapping tool that maps object-oriented database specifications to higher-order logic. M.Sc. Thesis, Universiteit Twente, August 1997.

[vK97] Maurice van Keulen. *Formal Operation Definition in Object-Oriented Databases*. PhD thesis, CTIT Ph.D. series, No. 97-16, University of Twente, Enschede, The Netherlands, 1997.

[VWP$^+$97] Jari Veijalainen, Jürgen Wäsch, Juha Puustjärvi, Henry Tirri, and Olli Pihlajamaa. Transaction Models in Cooperative Work—An Overview. In *Transaction Management Support for Cooperative Applications*, chapter 3, pages 27–58. Kluwer Academic Publishers, 1997.

[Wei88] William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[Wei93] William E. Weihl. The impact of recovery on concurrency control. *Journal of Computer and System Sciences*, 47:157–184, 1993.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[WK96] Jürgen Wäsch and Wolfgang Klas. History merging as a mechanism for concurrency control in cooperative environments. In *IEEE Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems*, pages 76–85, 1996.

Index

Samenvatting

Informatietechnologie is een steeds belangrijker rol gaan spelen in de samenleving. Veel dagelijkse handelingen, zoals het gebruik van een pincodeautomaat of een mobiele telefoon, worden mogelijk gemaakt door geautomatiseerde informatiesystemen. Gegevensbanken spelen daarbij niet zelden een belangrijke rol. Met deze toenemende invloed van informatietechnologie op de samenleving, is het van steeds groter belang dat er hulpmiddelen beschikbaar komen waarmee het ontwerp van een systeem op zijn juistheid kan worden gecontroleerd. Tot voor kort waren er echter nauwelijks hulpmiddelen om fouten in het ontwerp van een informatiesysteem op te sporen. Het afgelopen decennium is er aanzienlijke vooruitgang geboekt op dit terrein. Er zijn diverse krachtige verificatietechnieken ontwikkeld waarmee complexe analyses kunnen worden uitgevoerd op een hardware of een software ontwerp. Dit proefschrift gaat in op de vraag hoe één zo'n specifieke techniek, namelijk *theorem proving*, kan worden ingezet voor het analyseren van het ontwerp van een gegevensbanksysteem.

De verificatietechniek die wordt gepresenteerd in dit proefschrift gaat uit van het gebruik van een zogenaamde object-geörienteerde modelleringstaal. Dergelijke talen worden veel gebruikt voor het beschrijven van het ontwerp van gegevensbanksystemen. Voorgesteld wordt om verificatieondersteuning te ontwikkelen door middel van een representatie van het object-geörienteerde ontwerp van het gegevensbankschema in hogere order logica. Deze wiskundige representatie van een schema maakt het mogelijk om eigenschappen te bewijzen omtrent de correctheid van het ontwerp. Het hulpmiddel dat hierbij wordt gebruikt is een stellingenbewijzer ofwel *theorem prover*.

Het proefschrift beschrijft de vertaling van het object-geörienteerde schema naar een representatie in hogere order logica. Met name wordt daarbij ingegaan op de vraag hoe de karakteristieke object-geörienteerde aspecten van het schema kunnen worden gerepresenteerd in de logica. Op basis van de voorgestelde representatie wordt vervolgens beschreven hoe semi-automatische bewijsondersteuning kan worden verkregen, door gebruik te maken van een theorem prover. Uitgangspunt daarbij is een bestaand theorem prover systeem, namelijk Isabelle, dat gebruik maakt van de hogere order logica notatie.

Het proefschrift gaat nader in op een tweetal toepassingen van het verificatieraamwerk. In hoofdstuk 7 wordt een semi-automatische techniek beschreven die het mogelijk maakt om te bewijzen dat een wijziging van de toestand van de gegevensbank, door middel van de aanroep van een methode, in geen geval leidt tot een schending van de in het schema vastgelegde integriteitsregels. De voorgestelde techniek maakt gebruik van het Isabelle systeem, en is gebaseeerd op gangbare bewijstechnieken zoals term-herschrijving en deductie. De haalbaarheid van deze techniek wordt gedemonstreerd aan de hand van een case study. De experimenten tonen aan dat automatische verificatie in veel gevallen mogelijk is, en dat de tijd die nodig is om deze verificatie uit te voeren acceptabel is. In hoofdstuk 8 wordt een generalisatie van de verificatietechniek besproken. Hierbij wordt een toepassing beschreven in de context

van de zogenaamde *semantics-based transaction models*. Een veel voorkomende aanname in deze modellen is dat de ontwerper voor iedere methode m in het schema, een andere methode m^{-1} definiëert. Deze methode wordt intern gebruikt door het database systeem, om de effecten van een aanroep van m ongedaan te maken. De voorgestelde verificatietechniek biedt de mogelijkheid om te controleren of m gevolgd door m^{-1} inderdaad de database toestand ongewijzigd laat.

Acknowledgements

I would like to thank everyone who in some way contributed to this thesis. In particular I would like to thank Susan Even for her constant support and guidance. She spent a tremendous effort in providing feedback, and the many discussions we had highly influenced the contents of this thesis.

Of great help were the recommendations of Bart Jacobs and Herman Balsters, who provided various comments on earlier drafts of this thesis. My student Werner Toonk is acknowledged for starting up the implementation work discussed in Chapter 6. The results of his M.Sc. work ([Too97]) provided valuable insights for the direction in which this research evolved.

I want to thank all (former) members of the database group. I was lucky to have had Rolf de By as the supervisor of my M.Sc. project [Spe95]; this thesis is a continuation of that work. Maurice van Keulen is thanked for the discussions about update definition (see Section 3.3). Also I would like to mention Arjen de Vries; unforgettable is our stay at the EDBT Summer School 1997 on Capri.

Finally, I thank my family and Nidia for their support. My brother Peter inspired me to also pursue a Ph.D.