Validation Techniques for Object-Oriented Proof Outlines

SIKS Dissertation Series No. 2006-5

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 90-393-4217-2

Validation Techniques for Object-Oriented Proof Outlines

Validatietechnieken voor Objectgeoriënteerde Bewijsschetsen

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof. dr. W.H. Gispen,
ingevolge het besluit van het college voor promoties
in het openbaar te verdedigen
op woensdag 3 mei 2006 des middags te 2.30 uur

door

Cornelis Pierik

geboren op 20 november 1978, te Hasselt

promotor: prof. dr. J.-J. Ch. Meyer
co-promotor: dr. F.S. de Boer

Contents

Chapter 1

Introduction

Modern class-based object-oriented languages like Java and C# provide support for well-structured programs that reuse and extend existing framework classes. A well-designed class typically represents one essential piece of data and supplies a set of methods. Each of these methods implements a particular computational task which can be executed on the encapsulated data representation. Classes that are designed in this way often have elegant code that is amenable to formal analysis and verification.

Another factor that currently stimulates interest in the formal analysis of object-oriented software is the rapidly growing power of automated theorem provers. This development opens up the possibility to delegate proof construction steps in the verification process to such systems, which improves the reliability of these steps. For the use of theorem provers reduces the possibility that human errors render the outcomes of proofs invalid. But it is also of great importance for the scalability and cost-effectiveness of verification methods.

Proof construction is, however, only one aspect of software verification that may benefit from tool support. It is also possible to develop systems that compute and manage the proof obligations which ensure that a program satisfies certain properties. In recent years, several of these systems have been constructed. They typically differ in the input language that they support, the program properties which they establish, and the amount of user interaction that they require.

The verification technology with the highest degree of automation is probably *extended static checking* [FLL+02]. A static program verifier that performs extended static checking for annotated C# programs is currently being developed at Microsoft Research in Redmond [BLS05]. Similar tools (ESC/Java [FLL+02] and its successor ESC/Java2 [CK05]) have also been built in recent years for Java.

The aim of extended static checking is to move beyond ordinary type-checking towards checking for common programming errors like null dereferences and array bounds errors. By contrast, traditional program verification

tools try to prove full functional specifications. Such tools will only produce interesting results if the programmer has supplied a (more or less complete) functional specification for a piece of code, whereas an extended static checker also produces warnings for possible errors without a single line of specification. A specifier typically starts to write specification lines in response to the warnings that the checker generates. This process usually results in a specification that is sufficient to eliminate the warnings but that is not as strong as a full functional specification.

A closer look at both technologies reveals that there are also several similarities. Program verification for imperative programs is commonly based on Hoare logic [Hoa69]. The proof obligations for extended static checking are generated using verification condition generation techniques that also originated in the context of Hoare logic [FLL+02]. Moreover, the kind of specifications that extended static checkers eventually establish can also be proved using Hoare logics.

The main shortcoming of extended static checking is that it exploits a logic that is both *unsound* and *incomplete* [FLL+02]. This means in practice that the checker may produce less warnings than it should have produced, and that it may also generates some spurious warnings. This is usually defended by pointing out that an extended static checker must be cost-effective: every bug which the checker finds proves it usefulness, whereas a full analysis may require too many resources.

Where precisely the right balance between the required effort and the resulting safety assurances lies ultimately depends on the type of application. Certain applications must satisfy stronger safety properties than others. For example, the correctness of the software that controls an airplane is much more important than the correctness of the software behind an internet forum.

In this thesis we will develop a proof outline logic for object-oriented programs. This logic is both sound and complete. Thus it overcomes the limitations of extended static checking. But at the same time it is more suitable for automated verification than arbitrary Hoare logics because it defines a verification condition generation strategy that shows how the proof obligations of an annotated code fragment can be computed automatically. These proof obligations can then be passed to an automated theorem prover. This entire process can be implemented in a tool that is similar to an extended static checker.

Proof obligations which the automated theorem prover fails to prove should be examined in order to establish whether they indicate counter examples. Valid proof obligations which are beyond the power of an automated theorem prover can usually be proved using an interactive theorem prover.

In the following sections we will explain in more detail what a proof outline logic is. Moreover, we also outline some of the challenges that must be faced in order to obtain such a logic for object-oriented programming languages. The final section of this chapter contains an overview of this thesis.

1.1 Proof Outline Logics

Both extended static checking and our proof outline logic take proof outlines as input and translate these into a set of proof obligations. Proof outlines are annotated pieces of code. The annotation in proof outlines shows the specification that the code satisfies as well as important intermediate proof steps.

We will give an example proof outline below. But first we show the code without annotation. Our example is written in an object-oriented (Java-like) language. It concerns a method which can be invoked on an object in order to obtain a clone of this object. The clone is a fresh object of the same class as the receiver of the method. Moreover, it has the same values stored in its fields.

```
object clone() {
  Cloneable c;
  c := new Cloneable();
  c.field := this.field;
  return c;
}
```

The first line of the code declares the name (*clone*) and the return type (object) of the parameterless method; the code between braces that follows is the body of the method. The second line defines a local variable c of type *Cloneable*. The third line creates a new object of class *Cloneable* and assigns it to this local variable. We assume that each object of class *Cloneable* has one field (or instance variable) called *field*. The field of the new object gets the value of the field of the receiver of the object in the fourth line. The keyword this refers to the receiver of a method, and the expression this.*field* denotes the value of the receiver's field. Similarly, *c.field* denotes the value that is stored in the field of the object referenced by c. The last line reveals that the method returns the new object.

A proof outline of a method contains a description of the states in which the method may be called and a description of the states in which the method may terminate provided that its execution started in a permitted state. These descriptions are usually called the *precondition* and the *postcondition* of the method. They state what a method *requires* and what it *ensures*. Together, they form the *specification* of the method. We use logical formulas over program expressions to express these conditions.

We will also use logical formulas to describe *intermediate* states of method computations. These additional formulas characterize the states that may arise when control reaches particular points in the body of a method.

A proof outline always first lists the precondition and the postcondition of a method. These formulas are preceded by the keywords requires and ensures, respectively. Formulas that describe intermediate states are preceded by the keyword assert. Here is a proof outline of our *clone* method.

```
requires z.field = z';
ensures result.field = this.field ∧ ¬(result = z) ∧ z.field = z';
object clone() {
   Cloneable c;
   assert z.field = z';
   c := new Cloneable();
   assert ¬(c = z) ∧ z.field = z';
   c.field := this.field;
   assert c.field = this.field ∧ ¬(c = z) ∧ z.field = z';
   return c;
}
```

The symbols ¬ and ∧ in the proof outline denote the standard negation and conjunction operators. Other new elements in the proof outline are the logical variables z and z'. Such variables are used as placeholders for arbitrary values. A logical variable is a special kind of variable which may only be used in method annotations: it is not allowed to occur in the actual code of a method. Thus we know that the value of a logical variable never changes during a method execution.

The logical variables in the specification of our method give its clients some flexibility. If a client can show that the precondition holds in the initial state of the method execution for some particular pair of values for the logical variables z and z', then he gets the guarantee that the postcondition holds in the final state of that execution for the *same* values of these logical variables. Hence the clauses $z.field = z'$ in the precondition and the postcondition state that the value of the instance variable *field* of every object which existed when the method started is not modified during its execution. This holds despite the assignment $c.field := \mathsf{this}.field$ in the method body because at that point the local variable c references a fresh object which could not have been bound to the logical variable z in the initial state.

The keyword result in the postcondition denotes the result value. The first clause of the postcondition says that the result value (the clone) has the same value in its field as the receiver of the method. Its second clause $\neg(\mathsf{result} = z)$ states that the result value differs from the value of z. Since z can be any object that existed in the initial state, this clause effectively implies that the clone is a fresh object that has been allocated during the execution of the method.

Our proof outline also has formulas that describe the intermediate states. These formulas gradually grow as the building of the clone proceeds. The meanings of the clauses in these formulas are similar to the meanings of the corresponding clauses in the postcondition of the method.

This proof outline shows the kind of input that a proof outline logic (and an extended static checker) requires. The task of a proof outline logic is then to check whether the proof outline indeed leads to a valid proof for the specification of the method. A method specification is valid if every final state of a terminating computation of the method satisfies the postcondition. However,

this requirement only applies to computations which started in a state in which the precondition holds.

A proof outline logic consists of rules that specify a set of proof obligations for each proof outline. Such proof obligations are usually called *verification conditions*. A proof outline can be validated by showing that its verification conditions hold.

A typical feature of a proof outline logic is that its verifications conditions are similar to the formulas that are used in the proof outlines themselves. Consequently, the verification conditions of our proof outline logic will be logical formulas over program expressions.

The fact that the verification conditions of our logic are merely logical formulas is very important. For the complexity of the verification conditions determines how well theorem provers can automatically prove these proof obligations. More complex verification conditions will be harder to prove. Designing a dedicated proof procedure for a particular set of formulas is easier if the formulas in the set are more alike.

Specification Adaptation

What validation techniques does our proof outline logic provide to compute these verification conditions? In part, it is based on the same techniques as Hoare logics. For example, it exploits weakest precondition calculi for reasoning about assignments and object allocation. We will develop the required calculi by extending similar techniques in the work of De Boer [dB91, dB99] in order to handle inheritance and (subtype) polymorphism. The resulting calculi can be found in Chapter 4 and Chapter 6 of this thesis.

What distinguishes our proof outline logic from most Hoare logics is the fact that it does *not* use the standard Hoare rules for reasoning about method calls. Hoare initially proposed the following rule for reasoning about parameterless procedure calls [Hoa71].

$$\frac{\{P\}\ \mathsf{body}(p)\ \{Q\}}{\{P\}\ \mathsf{call}(p)\ \{Q\}}$$

This rule involves two *Hoare triples*. A Hoare triple has the form $\{P\}\ S\ \{Q\}$. This notation involves a statement S, a precondition P and a postcondition Q. A triple $\{P\}\ S\ \{Q\}$ indicates that every computation of S that starts in a state that satisfies P will only terminate in a state in which Q holds. The rule says that every specification which holds for the body of some procedure p (denoted by $\mathsf{body}(p)$) also holds for an arbitrary call to p.

Applying this basic rule in a proof outline logic would mean that every call may only be annotated with the same specification as the corresponding method. That is, the intermediate formula that precedes the call in the proof outline would have to match the precondition of the corresponding method, and the formula that follows the call would have to match its postcondition. It is most unlikely that these conditions can be satisfied in all circumstances.

A more flexible requirement would be that the formula that precedes the call only has to imply the precondition of the corresponding method. Similarly, we could require that the postcondition of the method implies the formula that follows the call. These two logical implications would then be added to the verification conditions of this proof outline. But these requirements are still too strong. One particular shortcoming of this solution is that it does not enable the client to exploit the logical variables in the method specification.

We will illustrate this shortcoming using an example that involves the specification of the *clone* method in our previous proof outline. Suppose that we want to prove the correctness of the following partial proof outline.

$$
\begin{array}{l}
\ldots \\
\mathsf{assert}\ \mathsf{true}; \\
v := \mathsf{this}.\mathit{clone}(); \\
\mathsf{assert}\ \neg(\mathsf{this} = v); \\
\ldots
\end{array}
$$

This formula that follows the call in this proof outline states that the object that is returned by the clone method is different from the cloned object. It should be possible to prove that this holds after the call since the specification of our *clone* method states that the clone is a fresh object. However, the formula true in this proof outline clearly does not imply the precondition $z.\mathit{field} = z'$ of the *clone* method, and the clause $\neg(\mathsf{result} = z)$ in the method's postcondition does not imply $\neg(\mathsf{this} = v)$. The last implication is partly invalid because we did not take the effect of the assignment of the result value to v into account. But even if we replace v by result in the clause $\neg(\mathsf{this} = v)$ we still have an invalid implication. The two implications are only valid if we additionally replace z by this and z' by $\mathsf{this}.\mathit{field}$ in the method specification.

This problem and some related problems are usually solved in Hoare logics by adopting several ad hoc substitution rules [Apt81]. Most existing Hoare logics for object-oriented languages also follow this approach [dB91, dB99, PHM99]. The main problem with these substitution rules is that it is not clear how they can be applied in the context of a proof outline logic. For they do not define a clear set of verification conditions for a method call.

Hoare himself was aware of some of the shortcomings of his basic procedure rule. He proposed an *adaptation rule* to overcome some of its limitations [Hoa71]. The purpose of an adaptation rule is to adapt existing specifications to what is required in some specific context. Recall that Hoare's initial rule requires that the specification of a call and the specification of the corresponding method be the same. With an adaptation rule one can simply adapt the method specification to ensure that its matches the specification of a call. This adaptation step is only possible if the premise of the adaptation rule holds, which suggests that this premise is a suitable verification condition for a proof outline logic.

One of the main contributions of this thesis is an adaptation rule for object-oriented languages. The basis of this rule is an adaptation rule for imperative

languages with global variables proposed by Olderog [Old83], which is superior to Hoare's original adaptation rule [Old83]. Designing an object-oriented adaptation rule is a non-trivial exercise because the state of an object-oriented program is far more complex than the simple mappings from variables to values in languages with global variables. The main problem is that its size cannot be determined *a priori*. For this is determined by the amount of objects that have been allocated during the execution of a program. We will say more about this issue in Section 1.2.

The specification adaptation techniques which the adaptation rule exploits also play a role in the context of *behavioral subtyping* [LW94]. There one must check whether the specification of an overriding method *refines* the specification of an overridden method. One way to answer this kind of question is to try to *adapt* the specification of the overriding method to that of the overridden method. There is a surprising similarity between this task and the original task of an adaptation rule. We take advantage of this insight in Chapter 8 of this thesis. There we reuse the validation techniques of our object-oriented adaptation rule to construct a test (a *specification match* [CC00]) that tells us whether a particular class is a behavioral subtype of some other class.

Abstraction Level

Another feature that distinguishes the proof outline logic in this thesis from other attempts to construct a logic for object-oriented programming languages is its *abstraction level*. The reasoning in our logic occurs as much as possible on the abstraction level of the source code. This becomes visible, for example, in the specifications that we allow in proof outlines. The formulas in these specifications are composed from the kind of expressions that are also used in the programming language. This ensures that they are easily intelligible to programmers.

Other Hoare-like logics for object-oriented programming incorporate explicit heap (or store) references in their specification language [PHM99, Mül02] or use a much stronger higher order language to specify their programs [vO01]. These extensions render the specifications in these languages less comprehensible. Moreover, the fact that our logic is both sound and complete (cf. Chapter 7) indicates that these extensions may well be inessential.

Besides ordinary program variables we also use logical variables in our specifications. We have already seen their usefulness in the context of the example proof outline on page 4. The types of the logical variables in specifications will often be normal program types but we will also use logical variables that denotes *finite sequences*. The usage of finite sequences in specifications can be traced back to the work of Tucker and Zucker [TZ88]. They needed sequences in their specification language to be able to express the strongest postconditions of statements that modify elements of abstract data types. Object reachability is another property that necessitates the use of finite sequences [Rot05]. We will additionally use sequences in our novel adaptation rule. The addition of

finite sequences has no major consequences for the abstraction level of most specification languages because these sequences are very similar to the arrays that are already present in most imperative languages.

Another aspect of the abstraction level of a program logic is the way it computes its verification condition. Several verification methodologies first translate a proof outline into some intermediate language before calculating the proof obligations. This intermediate language can be, for example, a guarded command language [FLL+02] or some other simple imperative language [MPMU04]. A disadvantage of this kind of indirection is that it makes the correspondence between a proof outline and its proof obligations more difficult to grasp. It also complicates the study of the formal properties of the underlying logic. The verification conditions of the proof outline logic in this thesis are computed directly from the annotated source code.

1.2 Object-Oriented Intricacies

This thesis does not focus on some particular object-oriented language. Instead, its aim is to provide proper reasoning techniques for a broad range of object-oriented features that are shared by several modern languages. The features that we discuss can all be found in, for example, Java and C#.

The remainder of this section explains some of the challenges of reasoning about object-oriented programs. Readers that are unfamiliar with object-oriented programming may first want to read Section 2.1, which contains an overview of the object-oriented terminology that is used in this thesis.

Field Shadowing

The two most prominent object-oriented features are probably *inheritance* and (subtype) *polymorphism*. Inheritance is the mechanism that enables code reuse in object-oriented languages. It is the reason behind the complex class hierarchies in object-oriented programs. A subclass inherits the fields and methods of its superclass.

Related to inheritance is the *field shadowing* phenomenon that is supported by both Java and C#. It occurs when a class declares a field with a name that is also the name of one of the fields that it inherits. Each instance of the new class then has several fields with the same identifier.

We will use an example to explain how most languages handle field shadowing. Let us assume that we have a class *Person* with a field *number* that contains the person's private telephone number. Now suppose that someone defines a subclass *Employee* of this class in which he also declares a field *number*. This second field is used to store the number of the telephone in the employee's office. In this way, each instance of class *Employee* has two fields that are both called *number*.

Recall that an expression $e.f$ has the value of the f field of the object denoted by e. In the presence of field shadowing it is no longer clear which field f is meant. In languages like Java and $C\#$, the (static) type of e determines which field is selected. For example, the expression $e.number$ denotes e's private number if e has type *Person*, and e's office number if e has type *Employee*.

It may seem a poor choice to create another field with the same identifier in a subclass. However, the designer of class *Employee* may have been unaware of the existence of the other field in class *Person*. After all, it is possible that this field is part of the hidden implementation details of that class. In practice, most language designers seem reluctant to constrain the freedom of subclass developers by preventing them from declaring fields with identifiers that have already been used in some superclass.

Field shadowing has important consequences for reasoning in object-oriented languages. For example, Hoare's classical axiom for reasoning about assignments [Hoa69] is no longer valid in the presence of field shadowing. Field shadowing makes the static types of expressions much more important. We further explain this observation in Section 4.1. In that chapter we also present techniques for coping with the notorious *aliasing* problem.

Dynamic Binding

A feature that is related to subtype polymorphism is *dynamic binding*. Subtype polymorphism allows an expression of a subtype to occur at places where normally an expression of a supertype is expected. This results in variables that reference values of subtypes. Consequently, expressions will sometimes denote values of subtypes of their static types.

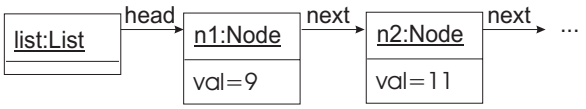
In particular, it leads to method calls in which the receiver is an object of some subclass of the type of the expression that denotes it. This class may also have a different implementation of the method that is invoked. Dynamic binding says that in such situations the class of the receiver determines which method is executed. However, this class usually remains unknown until we evaluate the receiver expression during the runtime execution of the method. Thus dynamic binding destroys the static connection between a method call and the executed method implementation.

This thesis proposes several adaptation rules for dynamically bound method calls. In Chapter 5 we present an adaptation rule that leads to a complete proof system for *closed* programs. In Chapter 8 we study the problem of reasoning about method calls in *open* (extensible) programs. For open programs techniques are needed that build proofs which cannot be invalidated by future program extensions. Section 8.2 presents two adaptation rules that satisfy this requirement. The first rule follows the standard *supertype abstraction* principle [LW90, LW95], but the second rule exploits a novel principle that results in a stronger rule.

Dynamic Object Allocation

Another factor that complicates many aspects of reasoning over object-oriented programs is the dynamic allocation of new objects. Every piece of code in an object-oriented program can contain statements that allocate new objects. The size of the state of an object-oriented program is therefore not fixed. Object-oriented languages share this property with all other languages with dynamically allocated variables that are referenced by pointers.

A data structure that is often encountered in languages with dynamically allocated variables is the linked list. The following pictures shows the initial part of the object structure of a linked list.



The leftmost block represents the list object, and the other blocks represent nodes in the list. The list object has a reference (depicted by means of an arrow) to the first node in the list, and each subsequent node has a reference to its successor. Each node in this particular list stores an integer value in its *val* field. The pictures shows only two nodes, but the number of nodes in a list is in principle unbounded.

Unbounded data structures are difficult to specify. Consider, for example, the task of specifying that the values in a linked list are sorted. We can use the formula $\mathsf{this}.\mathit{head}.\mathit{val} < \mathsf{this}.\mathit{head}.\mathit{next}.\mathit{val}$ to specify that the first two values in the list are in the right order. But what if there are more than two values in the list? Do we have to add a similar clause for each additional value? And how do we know how many values the list contains in a particular state?

We will use finite sequences to solve this kind of specification issues. We can, for example, use a logical variable L to represent the nodes in a linked list. With the formula $(\mathsf{this}.\mathit{head} = L[1]) \wedge (L[\mathsf{length}(L)].\mathit{next} = \mathsf{null})$ we can specify that the first object in L (denoted by $L[1]$) is the first node of L, and that the final object in L has no successor. We assume here that $\mathsf{length}(L)$ yields the length of the list. The expression null denotes the null reference. The formula $(\forall i \bullet 1 \leq i < \mathsf{length}(L) \rightarrow L[i].\mathit{next} = L[i+1])$ says that the objects in L form a linked list via their *next* fields. Finally, we can use the formula $(\forall i \bullet 1 \leq i < \mathsf{length}(L) \rightarrow L[i].\mathit{val} < L[i+1].\mathit{val})$ to express that the list is sorted.

The set of objects in an object-oriented state is also an unbounded structure. For there is no *a priori* bound on the number of objects in the state. This thesis shows how finite sequences can be used to model such states. We will exploit these techniques in the adaptation rules that we propose (see, e.g., Chapter 5) and in the completeness proof of our proof outline logic (Chapter 7).

There is also an interesting relation between quantification and object creation. Our specification language supports quantification over the set of existing objects. However, the set of existing objects is not fixed: it is extended each

time an object is created. This implies, for example, that the set of objects in the initial state of a method execution may differ from the set of objects in its final state. Consequently, one must be aware that a property that holds for all objects in some initial state does not automatically hold for all objects in the corresponding final state even when these objects have not been modified. For it is possible that in the meantime new objects which do not satisfy this property have been allocated. The adaptation rules in this thesis take this possibility into account.

Finally, there are also certain *invariants* that are falsifiable by object allocation. An invariant is a property that must hold in all *stable* states. There are different answers possible to the question which states are stable. We show how the invariant methodology [BDF+04] that is integrated in the Spec# programming system [BLS05] can be extended to support this type of invariant in Chapter 9.

1.3 Overview

In this section we explain the organization of this thesis and provide a more detailed description of its content. The first part of this thesis (Chapter 2-Chapter 7) defines a proof outline logic for *closed* object-oriented programs. Chapter 8 and Chapter 9 study techniques for reasoning about *open* programs. The final chapter of this thesis describes a verification tool that implements these techniques. Related work is discussed throughout this thesis.

Chapter 2 and Chapter 3 define an object-oriented programming language and its annotation, respectively. Chapter 2 starts with an overview of the object-oriented terminology that is used in this thesis. The rest of this chapter presents a sequential object-oriented language which embodies the features that are being studied in this thesis. It also describes an operational semantics for this language.

Chapter 3 introduces specification constructs that can be used to build proof outlines for programs that are written in this language. It also provides formal definitions of their meanings.

The next three chapters define verification condition generation techniques for the three kinds of basic statements in our object-oriented language: assignments, method calls, and creation statements. Chapter 4 defines weakest precondition calculi for assignments to local and instance variables. In these contexts we encounter the field shadowing phenomenon and the aliasing problem. The strongest postconditions of these statements are also studied in this chapter.

Chapter 5 covers the cornerstone of our proof outline logic. It first explains the shortcomings of the traditional rules for reasoning about method calls. Next, it introduces the techniques that are needed to build an object-oriented adaptation rule. The rule that is presented in this chapter defines the verification conditions of dynamically bound method calls in closed programs.

Chapter 6 shows how the verification conditions of creation statements can be computed. It defines a weakest precondition calculus for object allocation and the complementary strongest postconditions. It also reuses the techniques of the previous chapter to reason about constructor method calls.

Chapter 7 finishes the description of our proof outline logic for closed programs. It first outlines two verification condition generation strategies based on the techniques of the previous three chapters, and subsequently shows that at least one of these strategies defines a proof outline logic that is both sound and (relatively) complete.

The next chapter reveals how our proof outline logic can be transformed into a modular logic that is suitable for open programs. It first gives a formal definition of behavioral subtyping [LW94] in the context of our proof outlines. Next, it defines a novel specification match [CC00] that is based on the same techniques as our adaptation rules. This specification match can be used to determine whether a subclass is a behavioral subtype of its superclass. We also define two adaptation rules for modular programs in this chapter. Other contributions in this chapter are an object-oriented completeness notion for modular program logics and an analysis of several advanced specification constructs.

Chapter 9 studies a class of invariants that are falsifiable by object creation. It introduces *creation guards* to obtain a modular methodology that protects these invariants.

Finally, Chapter 10 describes a tool that implements our proof outline logic. It supports the building of proof outlines and automatically computes the corresponding verification conditions. Moreover, it interacts with a theorem prover in order to check these proof obligations.

Chapter 2

Object-Oriented Programming

Object-Oriented programming has existed for almost forty years now. For the pioneering work of Dahl and Nygaard on Simula started in the sixties of the previous century. Naturally, the notion of object-oriented programming evolved during the following decades. This thesis will be mostly concerned with the object-oriented features that are found in two relatively young object-oriented programming languages: Java and C#. They reveal in which direction object-oriented programming has developed, and which features have proved to be useful.

In the following section, we give an overview of the object-oriented features in these languages. The subsequent section presents an object-oriented language that embodies these features. We provide a formal description of its semantics and its type rules. The final section compares the introduced language with other object-oriented languages.

2.1 Object-Oriented Features and Terminology

The two central concepts in object-oriented programming are classes and objects. Classes are the structural units of object-oriented programs, whereas objects are the main data units of such programs. The two concepts are related: a class provides a blueprint of a specific set of objects.

An object can be viewed as a record with named fields. However, an object is more than a record because it has a unique identity. The fields of an object are often called its instance variables, and collectively they form the internal state of the object. Each object is an instance of a particular class; we will refer to this class as the dynamic type of the object; the dynamic type is sometimes also called the allocated type of an object, or its run-time type. Dynamic types should not be confused with ordinary (static) types, which are properties of

expressions.

A class is a syntactical entity that describes the structure of the internal state and the available functionality of its instances. Their internal state is defined by a set of field declarations. The operational information is given by a set of (instance) method declarations.

A method can be seen as a procedure with an implicit (additional) parameter that is always bound to an instance of the class in which it is declared. This object is called the receiver of the method invocation. Method invocations are usually referred to as method calls. The syntax of a method call specifies its intended receiver. Classes usually also contain a specific set of methods that are called constructors or constructor methods. Such methods are executed as the result of object creation, and their purpose is to ensure that the new object is immediately brought into a stable state.

Complex classes can be developed incrementally by declaring a class to be an extension of another class. The resulting class is said to be a subclass of the class that it extends. We call the latter its parent class, or its direct superclass. Thus each object-oriented program has a class hierarchy based on the corresponding subclass relation. We will assume in this thesis that the subclass relation of each program is a rooted tree. This means that each class, with the exception of the root class, has precisely one parent class.

Class extension relies on a mechanism that is called inheritance. It implies that each subclass inherits the fields and methods of its superclass. If a class has at most one superclass we speak of single inheritance. Some languages allow classes to have several superclasses, which enables situations in which the subclass inherits properties (fields and methods) from several classes. This feature is called multiple inheritance. Multiple inheritance requires a renaming mechanism to resolve name clashes between the properties of parent classes. We will not further discuss multiple inheritance in this thesis because it is not widely used.

The subclass may extend the superclass by defining new fields and methods. It is possible that the subclass defines fields with the same identifier as a field that it inherits. This situation is allowed in most object-oriented languages. The new fields is said to hide the inherited field. An instance of the new class will nevertheless have both fields. This phenomenon is called field shadowing.

A similar situation occurs if a class defines a method that it also inherits. In this case, we say that the new method overrides the inherited method. Method overriding can be exploited if it is combined with dynamic (or late) binding. Dynamic binding means that a method call results in the execution of the most specific method declared in (a superclass of) the dynamic type of the receiver. A method is more specific than another method if it is declared in a subclass of the other method's class. Method overriding combined with dynamic binding allows the programmer to change the behavior of instances of a subclass in comparison to instances of the superclass.

It is customary in object-oriented programming to define a (partial) *subtype* relation on the types of the programming language. Usually, this relation is

defined in such a way that every operation on the elements of a particular type t is also defined on the elements of every subtype t'. This convention enables a particular kind of polymorphism that is known as subtype polymorphism. Essentially, polymorphism is a relaxation of the typing restrictions. Without polymorphism, an assignment $x := e$ is only allowed if the type of the expression e and the type of the variable x are equal. Languages *with* subtype polymorphism also allow the assignment if the type of the expression is only a subtype of the type of the variable.

It is common to identify the subtype relation with the subclass relation. This means that every subclass is also a subtype. We will say more about this convention in Section 8.1.

2.2 An Object-Oriented Programming Language

In this section we describe a programming language (dubbed cOOre) that embodies the object-oriented features that we discussed in the previous section. The version presented here it a slightly extended version of its predecessors that either had no constructor methods [PdB03b] or were less explicit concerning type declarations [PdB05b].

We first describe the syntax of the languages. Its type restrictions are explained in Section 2.2.2. Section 2.2.3 contains a formal semantics of cOOre.

2.2.1 Syntax

The syntax of cOOre resembles that of Java, although we made some minor changes that should improve the readability. Figure 2.1 provides an overview of the syntax of the language.

A program π consists of a set of classes.

$$\pi \in \mathit{Prog} ::= \mathit{class}^*$$

Each valid program at least contains the root class object.

A class declaration specifies the (unique) name of the class, its parent class, a set of fields, a constructor method, and a set of instance methods.

$$\mathit{class} \in \mathit{Class} ::= \mathsf{class}\ C\ (\epsilon \mid \mathsf{extends}\ D)\ \{\ \mathit{field}^*\ \mathit{constr}\ \mathit{meth}^*\}$$

We use C, D, and E as typical elements of the set of class names. By ϵ we denote the empty sequence of tokens. A clause C extends D indicates that class C is an extension of class D. A class extends the root class object if the extends clause is omitted. Cycles in the subclass relation are not allowed.

A field declaration specifies its type t and its identifier.

$$\mathit{field} \in \mathit{Field} ::= t\ x\ ;$$

We typically use x to denote a field name. A class inherits all fields of its parent class. We allow classes to declare fields with names that equal those of inherited

$$
\begin{array}{rcl}
\pi \in \mathit{Prog} & ::= & \mathit{class}^* \\
\mathit{class} \in \mathit{Class} & ::= & \mathsf{class}\ C\ (\epsilon \mid \mathsf{extends}\ D)\ \{\ \mathit{field}^*\ \mathit{constr}\ \mathit{meth}^*\} \\
\mathit{field} \in \mathit{Field} & ::= & t\ x\ ; \\
t \in \mathit{Type} & ::= & \mathsf{int} \mid \mathsf{boolean} \mid C \mid D \mid E \mid \ \ldots \\
\mathit{constr} \in \mathit{Constr} & ::= & C(\bar{p})\{\ S\ \} \\
p \in \mathit{Parameter} & ::= & t\ u \\
\mathit{meth} \in \mathit{Meth} & ::= & \mathsf{void}\ m(\bar{p})\ \{\ S\ \} \mid t\ m(\bar{p})\ \{\ S\ \mathsf{return}\ e\ \} \\
S \in \mathit{Stat} & ::= & t\ u \mid u := e \mid e.x := e \mid S\ ;\ S \mid u := \mathsf{new}\ C(\bar{e}) \\
& \mid & e.m(\bar{e}) \mid u := e.m(\bar{e}) \mid \mathsf{if}\ (e)\ S\ \mathsf{else}\ S \mid \mathsf{while}\ (e)\ S \\
e \in \mathit{Expr} & ::= & \mathsf{null} \mid \mathsf{this} \mid u \mid e.x \mid (C)e \mid e\ \mathsf{instanceof}\ C \mid e\ ?\ e : e \\
& \mid & e = e \mid \mathsf{op}(\bar{e}) \\
\mathsf{op} \in \mathit{Op} & & \text{an arbitrary operator on elements of a primitive type}
\end{array}
$$

Figure 2.1: The syntax of cOOre.

fields, which leads to field shadowing (see Sect. 2.1). Thus objects may have multiple fields with the same name. However, within a class field names should be unique.

A type t is either a primitive type or a reference type.

$$t \in \mathit{Type} ::= \mathsf{int} \mid \mathsf{boolean} \mid C \mid D \mid E \mid \ \ldots$$

We will only consider the primitive types int and boolean. Reference types are simply class names in cOOre. Object fields of reference types may hold references to other objects. Thus objects can store references to other objects, which may result in complex reference networks that are usually called *object structures*.

A method declaration lists the return type of the method, its name m, a list of formal parameters $\bar{p}$, a statement S, and possibly a side-effect free expression e that denotes its return value.

$$\mathit{meth} \in \mathit{Meth} ::= \mathsf{void}\ m(\bar{p})\ \{\ S\ \} \mid t\ m(\bar{p})\ \{\ S\ \mathsf{return}\ e\}$$

A method with return type void returns no value, and it is therefore not allowed to have a result expression. The declaration of a method with a non-void return type must specify such an expression. A method name m should be unique in its class; we do not consider overloading for simplicity. By $\bar{p}$ we denote a comma-separated sequence of parameter declarations.

A parameter declaration specifies the type of the parameter and its identifier u.

$$p \in \mathit{Parameter} ::= t\ u$$

Note that our language does not support visibility modifiers. All fields and methods are considered to be public. Support for public (or protected) methods is a prerequisite for dynamic binding, for private methods cannot be overridden and are therefore always bound statically to an implementation.

Constructor method declarations are simplified method declarations.

$$\mathit{constr} \in \mathit{Constr} ::= C(\bar{p})\{\ S\ \}$$

Constructor methods always carry the name of the enclosing class. Their return type is always void and therefore omitted. Again, we prevent the complications of overloading by requiring that each class has only one constructor method.

We distinguish nine kinds of statements.

$$\begin{array}{rcl} S \in \mathit{Stat} & ::= & t\ u \mid u := e \mid e.x := e \mid S\ ;\ S \mid u := \mathsf{new}\ C(\bar{e}) \\ & \mid & e.m(\bar{e}) \mid u := e.m(\bar{e}) \mid \mathsf{if}\ (e)\ S\ \mathsf{else}\ S \mid \mathsf{while}\ (e)\ S \end{array}$$

A statement of the form $t\ u$ declares a new local variable u with type t. Local variables belong to a method and last as long as the method in which they are declared is being executed. Each local variable should be unique in its method, and it should be declared prior to its first use. A variable declaration additionally has an operational aspect: the variable receives its default value at the start of its lifetime.

We list assignments to local variables $u := e$ and assignments to fields of objects $e.x := e'$ separately because they require distinct reasoning techniques. An expression $e.x$ denotes field x of the object referenced by e. However, field shadowing requires us to be more precise. An expression $e.x$ always corresponds to the first field x that is found by an upward search of the class hierarchy starting in the static type of expression e.

A statement $u := \mathsf{new}\ C(\bar{e})$ allocates a new instance of class C and consequently calls the constructor method of class C with actual parameters $\bar{e}$. The new object is the receiver of the call. Afterwards, the local variable u becomes a reference to the new object. An invocation of method m with receiver e is denoted by $e.m(\bar{e})$. Here, $\bar{e}$ is a comma-separated list of expressions. The other statements are standard. We will sometimes put curly braces around statements to clarify their start and end.

All expressions are side-effect free.

$$\begin{array}{rcl} e \in \mathit{Expr} & ::= & \mathsf{null} \mid \mathsf{this} \mid u \mid e.x \mid (C)e \mid e\ \mathsf{instanceof}\ C \mid e\ ?\ e : e \\ & \mid & e = e \mid \mathsf{op}(\bar{e}) \end{array}$$

The literal null denotes the null reference. This value is the default value of any variable of a reference type. The keyword this always references the receiver of the active method.

An expression of the form $(C)e$ involve a cast, which changes the static type of the expression e to C. The value of $(C)e$ is normally the value of e, but its value is undefined if e is not a reference to an instance of (a subclass of) class C

or the null reference. We will say more about the use of casts in the following section in which we explain the role of types in the language.

The instanceof operator is used to obtain information regarding the run-time (allocated) type of an object. An expression e instanceof C is an expression of type boolean that is true if e references an instance of (some subclass of) class C or the null reference. An expression $e_1 \ ? \ e_2 : e_3$ is a conditional expression. The binary operator $=$ is the usual equality-operator which is denoted by $==$ in languages like Java and C#.

Finally, we also assume the existence of a set Op of operators on integers and booleans. This set includes, among others, the usual operators for addition and multiplication of integers, and boolean operators for conjunction, disjunction, and negation. For simplicity, we will assume that all operators are total. We additionally assume that Op includes literals like true, false, and the usual integer literals, which are seen as operators of arity 0.

2.2.2 Type Restrictions

Object-Oriented languages like Java and C# are strongly-typed, which means that each program written in such languages must obey a set of mathematically precise typing rules. The compiler typically infers a type for each expression and checks if each expression that occurs in a compound language construct has a type that is appropriate in that context. Typing rules are formulated such that a compiler is able to check statically if a program satisfies the rules.

Verification techniques such as Hoare logics or proof outline logics are similar to typing rules in two aspects. Firstly, they also aim to rule out programs with unwanted behavior. Secondly, they can also be applied statically, i.e., without actually running the program. The main difference between static verification techniques and type checking is in the amount of effort that is required from the programmer. Verification techniques require more program annotation, but they also ensure much stronger safety properties.

Static verification techniques are not independent of typing rules, however. They mostly assume at least type-safety of the language to which they are tailored. This means that any state that may arise during the execution of a program assigns values to the variables that are compatible with their declared types. The typing rules of a programming language are sound if they preserve this property. Moreover, they should ensure that each well-typed expression evaluates to a value that is compatible with its derived type.

The type-safety of Java has been the subject of several studies [NvO98, DEK99, IPW99]. Since cOOre is a subset of Java, we may safely assume that it is also type-safe, provided that our typing rules conform to the rules of Java. The proof outline logic that we develop in this thesis also assumes type-safety. We will explicate the typing rules that are typical for the object-oriented paradigm in the remainder of this section.

Typing in object-oriented language depends on the is-subtype-of relation between types. This relation in turn is based on the subclass relation, which

$$\frac{}{[\mathsf{null}] = \mathsf{NullT}} \qquad \frac{[e] = C \quad \mathsf{origin}(x, C) = C' \quad x \in \mathit{Fields}(C')}{[e.x] = [x]}$$

$$\frac{[e] \sim C}{[(C)e] = C} \qquad \frac{[e] \preceq C}{[e \ \mathsf{instanceof}\ C] = \mathsf{boolean}}$$

$$\frac{[e_1] = \mathsf{boolean} \quad [e_2] \preceq [e_3]}{[e_1 \ ?\ e_2 : e_3] = [e_3]} \qquad \frac{[e_1] = \mathsf{boolean} \quad [e_3] \preceq [e_2]}{[e_1 \ ?\ e_2 : e_3] = [e_2]}$$

$$\frac{[e_1] \sim [e_2]}{[e_1 = e_2] = \mathsf{boolean}}$$

Figure 2.2: Selected type rules for expressions. Here $\mathit{Fields}(C)$ denotes the set of fields that are declared in class C. The definition of origin is given in Section 2.2.3.

we denote by $\trianglelefteq$. It is the reflexive and transitive closure of the extends relation between classes. The null reference has a unique type NullT (the null type) that is a subtype of each reference type. The is-subtype-of relation of a program (written as $\preceq$) is the least set such that int $\preceq$ int, boolean $\preceq$ boolean, NullT $\preceq C$ for each class C, and $C \preceq D$ if $C \trianglelefteq D$. We write $t \prec t'$ to express that t is a proper subtype of t'. Type t is a proper subtype of t' if $t \preceq t'$ and $t \neq t'$.

We denote the type of an expression e by $[e]$. We have [null] = NullT. The value of [this] is always the reference type of the enclosing class. The type of an expression $e.x$ is the type of field x. We have $[(C)e] = C$. To ensure type-safety we require that $[e]$ is related to C. We say that two types t and t' are related, denoted by $t \sim t'$ if either $t \preceq t'$ or $t' \preceq t$. The same restriction applies to boolean expressions of the form e instanceof C.

The typing rules must ensure for each conditional expression $e_1 \ ?\ e_2 : e_3$ that $[e_1] =$ boolean, and that $[e_2]$ is related to $[e_3]$. The type of the conditional expression itself is $[e_2]$ if $[e_3] \preceq [e_2]$, and $[e_3]$ otherwise. An expression $e_1 = e_2$ is only valid if $[e_1]$ is related to $[e_2]$. The typing rules for operators on primitive expressions are standard. An overview of the most relevant type rules of expressions is given in Figure 2.2.

An expression new $C(\bar{e})$ has type C. The type of a method call $e.m(\bar{e})$ is the return type of the corresponding method. For both expressions we require that the types of the actual parameters $\bar{e}$ are subtypes of the types of the corresponding formal parameters $\bar{p}$. That is, it must be the case that $[e_i] \preceq [p_i]$, for each valid index i of the sequences $\bar{e}$ and $\bar{p}$. By $[p]$ we refer to the declared type of parameter p.

The type restrictions on actual parameters are the natural translation of the subtype polymorphism principle to parameter passing. The subtype polymorphism principle states that an expression of a subtype is acceptable whenever a

value of a supertype is expected. Therefore an assignment is allowed if the type of its right hand side is a subtype of the type of its left hand side. Similarly, an expression that denotes a return value is acceptable if its type is a subtype of the declared return type of the method.

Subtype polymorphism also plays a role in the context of method overriding. Naturally, a method that overrides a method declared in a superclass must have the same number of parameters. Furthermore, one could demand that its parameters have the same types as the parameters of the method that it overrides. However, a less rigid restriction suffices for type-safety.

It is common to speak of covariant and contravariant changes in types (see, e.g., [Bru02]). A covariant change occurs if a subclass uses a subtype compared to the the type used in the superclass. A contravariant change is a change in the opposite direction. Both types of changes can be permitted in overriding methods. A method that overrides another method may have a covariant change in its return type. That is, its return type may be a subtype of the return type of the method that it overrides. Parameter types may only vary contravariantly.

2.2.3 Formal Semantics

In this section we present an operational semantics for cOOre. A formal semantics is an essential prerequisite for rigorous results regarding properties of the language and the proof system that we present in this thesis.

It is common to distinguish *natural* operational semantics from *structural* operational semantics [NN92]. A natural (or big-step) semantics relates initial configurations with their final states. A configuration $\langle S, \sigma \rangle$ consists of a statement S and a state σ. Formulas in a natural semantics have the form $\langle S, \sigma \rangle \to \sigma'$. Such a formula states that a computation of S that starts in state σ terminates in state σ'.

Natural semantics have been derived from the structural operational semantics proposed by Plotkin [Plo81]. Plotkin's (small-step) semantics additionally contains rules of the form $\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$ that relates an initial configuration with its successor in the computation. Thus Plotkin's rules provide additional information about how the individual steps of the computations take place [NN92].

We will develop a natural semantics for cOOre because this simplifies some of the proofs. The additional information provided by a structural operational semantics will not be required. However, a small-step semantics may be a better guide for implementers of the language [DEK99, DVE00, vO01].

Variables

Programs in cOOre contain local variables and instance variables. We assume that *Var* denotes the set of all local variables of a program; it also includes the special-purpose variable this. By $Fields(C)$ we denote the set of fields that are

declared in class C. Let $\mathbb{C}$ be the set that contains the names of all classes declared in the program.

Due to inheritance objects may have several fields with the same identifier. As explained in Sect. 2.2.1, an expression $e.x$ always corresponds to the first declaration of a instance variable x as found by an upward search starting in class $[e]$. This search is formalized by the function

$$\mathsf{origin} : \mathbb{C} \times (\bigcup_{C \in \mathbb{C}} \mathit{Fields}(C)) \rightharpoonup \mathbb{C} \ ,$$

which yields the *origin* of field x of an object of class C.

$$\mathsf{origin}(C, x) = \begin{cases} C & \text{if } x \in \mathit{Fields}(C) \\ \mathsf{origin}(\mathsf{par}(C), x) & \text{otherwise} \end{cases}$$

Here $\mathsf{par}(C)$ denotes the parent class of class C; it is undefined for class object.

Values and Domains

We represent objects as follows. Each object has its own identity and belongs to a specific class. Let Id be an infinite set of object identities. The set of object values $\mathit{Val}(C)$ of a class $C \in \mathbb{C}$ is then given by $\{C\} \times \mathit{Id}$. The null reference *null* is the value of null.

By $\mathit{dom}(t)$ we denote the value domain of type t. The value domain of a reference type subsumes the object values of subclasses due to subtype polymorphism. It is defined by the following cases.

$$\begin{array}{rcl} \mathit{dom}(\mathsf{boolean}) & = & \{\mathit{tt}, \mathit{ff}\} \\ \mathit{dom}(\mathsf{int}) & = & \{\ldots, -1, 0, 1, \ldots\} \\ \mathit{dom}(\mathsf{NullT}) & = & \{\mathit{null}\} \\ \mathit{dom}(C) & = & \mathit{Val}(C) \cup (\bigcup_{t \in \{t' \mid t' \prec C\}} \mathit{dom}(t)) \end{array}$$

We finish this section with some simple results regarding values and domains.

Lemma 2.1. *If $t \preceq t'$ then $\mathit{dom}(t) \subseteq \mathit{dom}(t')$.*

Proof. A simple case analysis. □

Lemma 2.2. *For every class C we have*

$$\mathit{dom}(C) = (\bigcup_{D \in \{E \mid E \preceq C\}} \mathit{Val}(D)) \cup \{\mathit{null}\} \ .$$

Proof. A simple case analysis. □

Lemma 2.3. *If $C \not\sim D$ then $\mathit{dom}(C) \cap \mathit{dom}(D) = \{\mathit{null}\}$.*

Proof. Recall that $dom(\mathsf{NullT}) = \{null\}$ and that $\mathsf{NullT} \preceq C$, for every reference type C. Then by Lemma 2.1 we have $null \in dom(C)$ for every C. Hence $null \in dom(C) \cap dom(D)$ and $\{null\} \subseteq dom(C) \cap dom(D)$

Secondly, we will prove that $dom(C) \cap dom(D) \subseteq \{null\}$ by showing that $o \in dom(C) \cap dom(D)$ leads to a contradiction if $o \neq null$. So let $o = (E, i) \in dom(C) \cap dom(D)$. Hence $o \in dom(C)$ and $o \in dom(D)$. Then $E \preceq C$ and $E \preceq D$ by Lemma 2.2. In other words, E must be a subclass of C and of D. The superclasses of E form a chain, so either C is a subclass of D or D is a subclass of C. Therefore $C \sim D$, which contradicts the assumption $C \not\sim D$. □

States

States of object-oriented programs consist of a stack which assigns values to local variables, and a heap (or object store) that contains all allocated objects and the values of their fields.

Only the top of the stack, which assigns values to the local variables and the parameters of the active method, is relevant in the semantics. For this reason, we model a stack $s \in \mathit{Stacks}$ as a total function of local variables to values. The function represents the top of the (actual) stack. Formally, *Stacks* is the set

$$\prod_{u \in Var} dom([u]) \ .$$

Note that we write $\prod_{a \in A}(P(a))$ to denote a (generalized) cartesian product. An element of this set is a function that assigns to every element $a \in A$ an element of the set $P(a)$.

A heap maps each existing object to its internal state. The internal state of an object is a mapping from fields to values. The internal state of an instance of class C assigns values to all inherited fields and to all fields declared in class C. By $\mathit{IntSts}(C)$ we denote the set of internal states of C-Objects. We have

$$\mathit{IntSts}(C) = \Big(\prod_{D \in \{E \mid C \preceq E\}} \Big(\prod_{x \in Fields(D)} dom([x]) \Big) \Big) \ .$$

A heap $h \in \mathit{Heaps}$ is a partial function that maps each *existing* object to its internal state. The set *Heaps* is defined as follows.

$$\mathit{Heaps} = \prod_{C \in \mathbb{C}} \Big(\mathit{Id} \rightharpoonup \mathit{IntSts}(C) \Big).$$

We say that an object o exists in heap h if $h(o)$ is defined. The domain of a heap h, denoted by $dom(h)$, is the set of objects o such that $h(o)$ is defined extended with *null*.

As a result of the above definitions, $h(o)$ is the internal state of an object $o = (C, id)$, if it exists. The value of some field x declared in class C is given by $h(o)(C)(x)$, for every object o in the value domain of class C.

Not every heap is valid. We say that a heap h is valid if all instance variables of objects that exist in h either reference *null* or an object that exists in h. In other words, valid heaps do not have dangling references. Likewise, a stack s is said to be consistent with heap h if all variables either reference an object that exists in h or *null*. A state (s, h) is valid if h is valid and s is consistent with h. We will silently assume that all considered states are valid in the remainder of this thesis.

Expression Evaluation

Expressions in COORE have no side effects. Evaluation of expressions will therefore result in a value; the state will always remain unchanged. We use the following evaluation function for expressions.

$$\mathcal{E}[\![\_]\!] : \mathit{Expr} \times (\mathit{Stacks} \times \mathit{Heaps}) \to \mathit{Val}$$

Here *Val* denotes the set $(\bigcup_t \mathit{dom}(t)) \cup \{\bot\}$. The value $\bot$ stands for 'undefined'. Expressions can be undefined due to dereferencing of the null pointer or illegal casts.

The definition of $\mathcal{E}[\![e]\!](s, h)$ can be found in Figure 2.2.3. Note that the definition of $\mathcal{E}[\![\mathsf{op}(e_1, \ldots, e_n)]\!](s, h)$ rules out non-strict (short-circuit) operators that do not always evaluate all their operands, which implies that their value need not be undefined if the value of one of their operands is undefined. This is not an essential restriction; the proof system that we present is also suitable for languages with non-strict operators.

The following lemma states that expression evaluation is type-safe.

Lemma 2.4. *For every expression e and state (s, h) we have*

$$\mathcal{E}[\![e]\!](s, h) \in \mathit{dom}([e]) \cup \{\bot\} \ .$$

Proof. By structural induction on e. □

The Transition Relation

By $\langle S, (s, h)\rangle \to (s', h')$ we denote that a computation of S that starts in state (s, h) ends in state (s', h').

We will discuss some interesting cases in detail. We first consider the semantics of local variable allocation. Fresh local variables have their standard default values after allocation. The default value of a variable depends on its type. We assume that $\mathsf{init}(t)$ denotes the default value of type t according to the following cases.

$$\begin{aligned} \mathsf{init}(\mathsf{boolean}) &= \mathit{ff} \\ \mathsf{init}(\mathsf{int}) &= 0 \\ \mathsf{init}(C) &= \mathit{null} \end{aligned}$$

Local variable allocation then proceeds as described by the following rule.

$$\frac{}{\langle t\ u\ , (s, h)\rangle \to (s[u \mapsto \mathsf{init}([u])], h)} \quad (\mathit{VA})$$

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathsf{null}]\!](s,h) & = & \mathit{null} \\
\mathcal{E}[\![\mathsf{this}]\!](s,h) & = & s(\mathsf{this}) \\
\mathcal{E}[\![u]\!](s,h) & = & s(u) \\
\mathcal{E}[\![e.x]\!](s,h) & = & \begin{cases} \bot & \text{if } v \in \{\mathit{null}, \bot\} \\ h(v)(\mathsf{origin}([e],x))(x) & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\ (C)e\ ]\!](s,h) & = & \begin{cases} v & \text{if } v \in \mathit{dom}(C) \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{E}[\![e \text{ instanceof } C]\!](s,h) & = & \begin{cases} \bot & \text{if } v = \bot \\ \mathit{tt} & \text{if } v \in \mathit{dom}(C) \\ \mathit{ff} & \text{otherwise} \end{cases} \\
\mathcal{E}[\![e_1 \ ?\ e_2 : e_3]\!](s,h) & = & \begin{cases} \bot & \text{if } v_1 = \bot \\ v_2 & \text{if } v_1 = \mathit{tt} \\ v_3 & \text{if } v_1 = \mathit{ff} \end{cases} \\
\mathcal{E}[\![e_1 = e_2]\!](s,h) & = & \begin{cases} \mathit{tt} & \text{if } v_1 = v_2 \\ \mathit{ff} & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\mathsf{op}(e_1,\ldots,e_n)]\!](s,h) & = & \begin{cases} \bot & \text{if } v_i = \bot \\ \bar{\mathsf{op}}(v_1,\ldots,v_n) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2.3: Evaluation of expressions. In the left hand sides we assume that v and v_i, with $i \in \{1, 2, \ldots\}$, abbreviate $\mathcal{E}[\![e]\!](s,h)$ and $\mathcal{E}[\![e_i]\!](s,h)$, respectively. By $\bar{\mathsf{op}}$ we denote the fixed interpretation of operator op.

We write $s[u_1 \ldots u_n \mapsto v_1, \ldots, v_n]$ to denote the stack that maps u_i to v_i, for $i \in \{1 \ldots n\}$, and that assigns all other variables the same value as s. Note that local variable allocation always succeeds, for this rule has no antecedent.

Assignments to local variables of the form $u := e$ only succeed if the value of expression e is defined.

$$\frac{\mathcal{E}[\![e]\!](s,h) \neq \bot}{\langle u := e\ , (s,h)\rangle \rightarrow (s[u \mapsto \mathcal{E}[\![e]\!](s,h)], h)} \qquad (LA)$$

Field assignments $e.x := e'$ only terminate if the values of the expressions e and e' are defined. The rule assumes that field x is defined in class D. The new heap $h[o.x_D \mapsto v]$ is obtained from h by assigning the value of e' to field x of class D of the object referenced by e.

$$\frac{\mathcal{E}[\![e]\!](s,h) = o \notin \{\bot, \mathit{null}\} \quad \mathcal{E}[\![e']\!](s,h) = v \neq \bot \quad \mathsf{origin}([e],x) = D}{\langle e.x := e'\ , (s,h)\rangle \rightarrow (s, h[o.x_D \mapsto v])} \qquad (FA)$$

Here $h' \equiv h[o.x_D \mapsto v]$ denotes the heap h' such that $h'(o)(D)(x) = v$, and h' equals h in all other cases.

Next, we discuss the semantics of method calls. Let us consider a call to method m on object e_0 of the form $u := e_0.m(e_1, \ldots, e_n)$. Recall that method calls are bound dynamically to a method implementation. Therefore the value

of e_0 is evaluated first. The call fails if the value of e_0 is undefined. Let $\mathcal{E}[\![e_0]\!](s,h) = (C, id)$. Then this call is bound to the implementation of method m that is declared in class C or otherwise the implementation that C inherits from its parent class. We denote this method implementation by $\mathsf{meth}(C, m)$.

$$\frac{\begin{array}{c}\mathcal{E}[\![e_0]\!](s,h) = o = (C, id)\\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for every } i \in \{1 \ldots n\}\\ \mathsf{meth}(C,m) \equiv t\ m(p_1, \ldots, p_n)\{\ S\ \mathsf{return}\ e\ \}\\ \langle S, (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h)\rangle \to (s', h')\\ \mathcal{E}[\![e]\!](s', h') = v \neq \bot\end{array}}{\langle u := e_0.m(e_1, \ldots, e_n), (s,h)\rangle \to (s[u \mapsto v], h')} \qquad (MC_1)$$

The call starts with the context switch in which the value of e_0 is assigned to this and the values of the actual parameters $\bar{e} = e_1, \ldots, e_n$ are assigned to the formal parameters $\bar{p} = p_1, \ldots, p_n$. We identify p_i with its actual declaration here, which has the form $t_i\ u_i$. It is the parameter u_i that is updated in the context switch. After execution of the body S the value of e is assigned to u. Possible updates of the stack s by the body S are then discarded, but changes to the heap remain intact when the method returns.

We have a similar rule for calls to methods that do not return a value.

$$\frac{\begin{array}{c}\mathcal{E}[\![e_0]\!](s,h) = o = (C, id)\\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for every } i \in \{1 \ldots n\}\\ \mathsf{meth}(C,m) \equiv \mathsf{void}\ m(p_1, \ldots, p_n)\{\ S\ \}\\ \langle S, (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h)\rangle \to (s', h')\end{array}}{\langle e_0.m(e_1, \ldots, e_n), (s,h)\rangle \to (s, h')} \qquad (MC_2)$$

An assignment $u := \mathsf{new}\ C(e_1, \ldots, e_n)$ involves the allocation of a new object, and the execution of the constructor method of class C. The instance variables of the new object initially have their default values. Let $\mathit{init}(C)$ be the initial internal state of an object of class C. We have $\mathit{init}(C)(D)(x) = \mathsf{init}([x])$ for every field $x \in \mathit{Fields}(D)$ declared in some class D such that $C \trianglelefteq D$. It is undefined for all other fields.

By $h \cdot [o \mapsto \mathit{init}(C)]$ we denote the heap that is obtained from h by extending its domain with an object o such that $(h \cdot [o \mapsto \mathit{init}(C)])(o) = \mathit{init}(C)$. It is only defined if object o is fresh, i.e., if $h(o)$ is undefined.

Let $\mathsf{constr}(C)$ be the constructor method declaration in class C. The rule below then describes object creation.

$$\frac{\begin{array}{c}o = (C, id) \quad h(o) \text{ is undefined}\\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for every } i \in \{1 \ldots n\}\\ \mathsf{constr}(C) \equiv C(p_1, \ldots, p_n)\ \{\ S\ \}\\ \langle S\ , (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h \cdot [o \mapsto \mathit{init}(C)])\rangle \to (s', h')\end{array}}{\langle u := \mathsf{new}\ C(e_1, \ldots, e_n)\ , (s,h)\rangle \to (s[u \mapsto o], h')} \qquad (OC)$$

The (standard) rules for the other statements are listed in Figure 2.4.

$$\frac{\langle S_1,(s,h)\rangle \to (s'',h'') \quad \langle S_2,(s'',h'')\rangle \to (s',h')}{\langle S_1;\ S_2,(s,h)\rangle \to (s',h')} \qquad (SC)$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = tt \quad \langle S_1,(s,h)\rangle \to (s',h')}{\langle \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2,(s,h)\rangle \to (s',h')} \qquad (If_1)$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = ff \quad \langle S_2,(s,h)\rangle \to (s',h')}{\langle \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2,(s,h)\rangle \to (s',h')} \qquad (If_2)$$

$$\frac{\langle \mathsf{while}\ (e)\ S,(s,h) \xrightarrow{n} (s,h) \text{ for some } n \in \{0,1,\ldots\}}{\langle \mathsf{while}\ (e)\ S,(s,h)\rangle \to (s,h)} \qquad (Wh)$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = ff}{\langle \mathsf{while}\ (e)\ S,(s,h)\rangle \xrightarrow{0} (s,h)} \qquad (Wh_1)$$

$$\frac{\begin{array}{c}\mathcal{E}[\![e]\!](s,h) = tt \\ \langle S,(s,h)\rangle \to (s'',h'') \\ \langle \mathsf{while}\ (e)\ S,(s'',h'')\rangle \xrightarrow{n} (s',h')\end{array}}{\langle \mathsf{while}\ (e)\ S,(s,h)\rangle \xrightarrow{n+1} (s',h')} \qquad (Wh_2)$$

Figure 2.4: Additional rules of the operational semantics.

2.3 Related Languages

The language COORE is supposed to model the object-oriented core of languages like Java [GJSB00] and C# [Mok03]. This entails that we left out many features of Java and C# that did not seem relevant to the subject of this thesis, but that are certainly useful to programmers.

The most important omission is concurrency. Java and C# both have a thread class that can be instantiated to obtain another execution thread. Ábrahám et al. have developed a proof outline logic for a multi-threaded subset of Java [AMdBdRS02, AdBdRS03, Á05]. Their work does not deal with inheritance and subtype polymorphism.

Our language does not have an error handling mechanism such as the exceptions in Java and C#. However, exception handling is not a characteristic feature of the object-oriented paradigm. Hoare-like logics that deal with exceptions have been proposed by several authors [HJ00, Hui01, vO01]. Kowaltowski wrote an early work on handling side effects and jumps in Hoare logics [Kow77]. Throwing an exception can be seen as a particular kind of jump. Kowaltowski's

work is also relevant for reasoning about side effects in Java and C#-expressions. Nevertheless, we omitted expressions with side effects in cOOre for clarity. We also left out interfaces and abstract classes, which do not seem to pose interesting problems from a proof-theoretical perspective.

Both Java and C# have visibility (or accessibility) modifiers that control which class members are visible outside a class. There are subtle differences between Java and C# in the meaning of modifiers [Mok03], and both languages have at least one visibility mode that is not supported by the other language. We simply assume that all class members have public visibility in cOOre, i.e., they can always be accessed in other classes. We will indicate the consequences of this choice for the proof system in the various chapters that describe its constituents.

In Chapter 9 we will extend cOOre with static variables. It is not necessary to add static variables already because reasoning about static variables is similar to reasoning about local variables. The main difference is their initialization process: static variables may be initialized in Java by code that is executed during class initialization. It is difficult to determine when a class is actually initialized. The general rule is that a class is initialized prior to the first use of one of its members or prior to the first allocation of its first instance. However, the Java Language Specification also has a disturbingly vague sentence that says "Invocation of certain reflective methods in class `Class` and in package `java.lang.reflect` also causes class or interface initialization" [GJSB00, p. 237]. A second complication is that the textual order of static variable declarations influences the outcome of class initialization [GJSB00, JKW03]. Class initialization in C# proceeds along the same lines [Mok03], but there may well be subtle differences. For these reasons, it is essential to put further restrictions on class initialization in order to facilitate formal reasoning. However, this falls outside the scope of this thesis.

Java has a complex inference system that ensures that each local variable is explicitly assigned a value before its first use. Thus the initial default value of a local variable becomes irrelevant. We did not model this aspect of Java, but it is certainly compatible with our model.

Object-Oriented languages that are older than Java (and certainly older than C#) sometimes have features that are not incorporated in later object-oriented languages. For example, languages like C++ and Eiffel [Mey92] support multiple inheritance; Smalltalk only supports single inheritance [GR89]. The main reason for its omission in modern object-oriented languages seems to be their designers' desire to aim for simplicity.

Many other compact subsets of Java have have been proposed. The most radical subset is, presumably, Featherweight Java [IPW99], which even eliminates assignments; its main goal was to make its proof of type soundness as concise as possible. Other sequential subsets include Java-K [PHM99] and Java^{light} [vO01], of which the latter is the most substantial subset. The design goal behind Java$_{\mathrm{MT}}$ [AMdBdRS02, AdBdRS03] was to model Java's concurrency mechanism.

Chapter 3

Program Annotation

This chapter introduces several key ingredients of the formal system that we study in this thesis. First of all, it provides a formal description of *interface specifications*. An interface specification of a method will consist of a precondition and a postcondition. Both these formulas will be members of the formal specification language that we describe in the following section.

The main subject of this thesis are *proof outlines*. Proof outlines are a means to give a justification for the interface specification of a method. Proof outlines are annotated programs that satisfy certain validity constraints. We introduce the syntax and semantics of annotated programs in this chapter. Related work on specification languages is discussed in the last section.

3.1 A Specification Language

Our proof outline logic is based on an assertion language for cOOre, dubbed cOOral, that is tailored to the programming language. For example, most expressions in cOOral are simply programming language expressions. We add only a few additional language constructs to enhance the expressiveness of the language. Our reluctance to add additional constructs to the language is motivated by our desire to work with a specification language that can be easily understood by the average programmer. The Java specification language JML [LBR04] is a similar attempt to design a specification language that is closely tailored to a programming language.

The first necessary addition is *logical* variables. Logical variables are also called *freeze* or *ghost* variables. We will use logical variables as bound variables of quantifiers. It is also possible to use logical variables to relate initial and final values of variables in specifications. For example, a Hoare triple

$$\{x = z\}\ x := x + 1\ \{x = z + 1\}$$

uses the logical variable z to freeze the initial value of program variable x in the precondition $x = z$. The postcondition $x = z + 1$ then reveals that x has been

incremented by one. However, our assertion language will also allow expressions of the form $\mathsf{old}(e)$, which denote the value of expression e in the initial state of a method execution.

A final extension concerns the set of valid types in the assertion language, which we denote by $\mathcal{T}$. This set extends the set of programming language types to enable quantification over *finite* sequences of state elements. Thus logical variables may also denote sequences. We write t^* to denote the sort of finite sequences of elements from the domain of t. Recall from the previous chapter that *Type* is the set of types in the programming language. The set of types $\mathcal{T}$ of the assertion language is defined by $\mathcal{T} = \mathit{Type} \cup \{t^* \mid t \in \mathit{Type}\}$. We will assume that t ranges over the latter set in the assertion language.

The set of logical expressions *PExpr*, with typical element p, is defined as follows.

$$\begin{array}{rcl} p \in \mathit{PExpr} & ::= & \mathsf{null} \mid \mathsf{this} \mid u \mid p.x \mid (C)p \\ & \mid & p \ \mathsf{instanceof}\ C \mid p\ ?\ p : p \mid p = p \mid \mathsf{op}(\bar{p}) \\ & \mid & z \mid z[p] \mid \mathsf{length}(z) \mid \mathsf{undefined} \mid \mathsf{defined}(p) \end{array}$$

Note that the first two lines are copied from the grammar of programming language expressions; the last line shows five new cases.

We use z as a typical element of the set *LVar* of logical variables. As explained above, its static type $[z]$ is an element of the set $\mathcal{T}$, which implies that a logical variable may also denote a sequence of values. Let z be a logical variable of some sequence type. Then $z[p]$ denotes the element at index p in the sequence z, and $\mathsf{length}(z)$ denotes the length of the sequence. Valid indices range from 0 to $\mathsf{length}(z) - 1$. An empty sequence has length 0. These two operators will be the only valid operators on sequences.

The keyword $\mathsf{undefined}$ has the value $\bot$; An expression $\mathsf{defined}(p)$ can be used to test whether the value of p is defined, i.e., whether it is different from $\bot$, in a particular state. Note that an expression $\mathsf{defined}(p)$ is *not* equivalent to $\neg(p = \mathsf{undefined})$ (where $\neg$ denotes the logical negation operator) because the value of an expression $p_1 = p_2$ is $\bot$ if either p_1 or p_2 has the value $\bot$.

Observe that the grammar of the set of logical expressions *PExpr* does not include expressions of the form $\mathsf{old}(e)$. Actually, *PExpr* is the set of expressions that are valid in preconditions. An expression of the form $\mathsf{old}(e)$, which has the value of e in the initial state of a method execution, does not make sense in a precondition P because preconditions are always evaluated in the initial state.

For this reason, we introduce a second set *QExpr*, which is the set of arbitrary logical expressions. The grammar for this latter set is obtained from the grammar of *PExpr* by adding the clause $\mathsf{old}(e)$, which results in the following grammar.

$$\begin{array}{rcl} q \in \mathit{QExpr} & ::= & \mathsf{null} \mid \mathsf{this} \mid u \mid q.x \mid (C)q \\ & \mid & q \ \mathsf{instanceof}\ C \mid q\ ?\ q : q \mid q = q \mid \mathsf{op}(\bar{q}) \\ & \mid & z \mid \mathsf{old}(e) \mid z[q] \mid \mathsf{length}(z) \mid \mathsf{undefined} \mid \mathsf{defined}(q) \end{array}$$

A program expression e in an expression of the form $\mathsf{old}(e)$ may refer to the parameters of a method (including the implicit parameter this), and to fields of objects that are reachable from the parameters. It is not allowed to refer to local variables other than the parameters of the method because such variables are not allocated in the initial state.

Assertion language formulas are built from logical expressions in the usual way.

$$Q \in \mathit{QForm} ::= q \mid \neg Q \mid Q \wedge Q \mid (\exists z : t \bullet Q)$$

The set of preconditions PForm, with typical element P, is build from precondition expressions p in the same way.

Other boolean connectives can be encoded as usual. For example, we have $P_1 \vee P_1 \equiv \neg(\neg P_1 \wedge \neg P_2)$, and $P_1 \rightarrow P_2 \equiv \neg(P_1 \wedge \neg P_2)$.

We write $(\exists z : C \bullet Q)$ to express that Q holds for some *existing* object of class C. Similarly, $(\exists z : C^* \bullet Q)$ holds if Q holds for some sequence of existing objects from $\mathit{dom}(C)$. We sometimes omit the type t in a formula $(\exists z : t \bullet Q)$ if it is clear from the context. A formula of the form $(\forall z \bullet Q)$ abbreviates the formula $\neg(\exists z \bullet \neg Q)$. We also use two other useful abbreviations. A formula $z \in z'$ will stand for $(\exists i \bullet 0 \leq i < \mathsf{length}(z') \wedge z = z'[i])$, and $z \sqsubseteq z'$ abbreviates the formula $(\forall i \bullet 0 \leq i < \mathsf{length}(z) \rightarrow z[i] \in z')$.

Quantification over finite sequences is a powerful extension of the language. It allows us, for example, to express reachability via a reference chain of arbitrary length in the assertion language, as witnessed by the following example.

Example 3.1. *Let Node be a class with fields next* : *Node and val* : int. *A second class List with field fst* : *Node can be used to build a linked list of elements of class Node, with fst referencing the head of the list. The formula*

$$\begin{aligned}(\exists z : \mathit{Node}^* \bullet\ & z[0] = \mathsf{this}.\mathit{fst} \wedge z[\mathsf{length}(z) - 1] = n \wedge \\ & (\forall i \bullet 0 \leq i < \mathsf{length}(z) - 1 \rightarrow (z[i] \neq \mathsf{null} \wedge z[i].\mathit{next} = z[i+1]))\end{aligned}$$

holds if Node n can be reached via a finite chain of next fields from the head of the list. Let $\mathsf{reachable}(n)$ *abbreviate the above formula. A value v occurs in the list if*

$$(\exists n : \mathit{Node} \bullet n \neq \mathsf{null} \wedge \mathsf{reachable}(n) \wedge n.\mathit{val} = v) \ .$$

3.1.1 Formal Semantics

The typing rules for logical expressions are similar to those for program expressions. In our assertion language, we use a new type undefT, which is the type of the keyword $\mathsf{undefined}$.

In the assertion language, we define $\preceq$ to be the least set such that $\mathsf{int} \preceq \mathsf{int}$, $\mathsf{boolean} \preceq \mathsf{boolean}$, $\mathsf{NullT} \preceq C$ for each class C, and $C \preceq D$ if $C \trianglelefteq D$; moreover, $\mathsf{undefT} \preceq t$ if t is a primitive type or a reference type C.

We do not extend the subtype relation to sequence types. As a result, an expression of the form $e = e'$ is not well-typed if $[e]$ or $[e']$ is a sequence type

$$\frac{[e] = t}{[\mathsf{old}(e)] = t} \qquad \frac{[z] = t^* \quad [q] = \mathsf{int}}{[z[q]] = t} \qquad \frac{[z] = t^*}{[\mathsf{length}(z)] = \mathsf{int}}$$

$$\frac{}{[\mathsf{undefined}] = \mathsf{undefT}} \qquad \frac{[q] = t}{[\mathsf{defined}(q)] = \mathsf{boolean}}$$

Figure 3.1: Typing rules of logical expressions (additional cases).

(cf. the rule for this type of expression in Fig. 2.2). Sequences are dissimilar from objects in one aspect: they have no identity. However, structural equality of two sequences z_1 and z_2 can be expressed in the assertion language by means of a formula

$$\mathsf{length}(z_1) = \mathsf{length}(z_2) \wedge (\forall i \bullet 0 \leq i < \mathsf{length}(z_1) \rightarrow z_1[i] = z_2[i]) \ .$$

The new type rules are listed in Fig. 3.1. The type rules for assertions are straightforward and therefore omitted; it suffices to remark that only logical expressions of type boolean qualify as basic formulas.

In order to evaluate formulas from the assertion languages we must slightly extend our state model. First, we must define the domains of sequence types. We view lists as pairs (f, n) of a function f that maps indices to values, and a natural number $n \in \mathbb{N}$ that represents the length of the list. Let t be some type of the programming language. Then we define

$$dom(t^*) = \bigcup_{n \in \mathbb{N}} \{(f, n) | f \in \prod_{i=0}^{n-1} dom(t)\} \ .$$

For a sequence $\bar{v} = f(n)$ we define $rng(\bar{v}) = \{f(i) | 0 \leq i < n\}$.

Secondly, we extend the domains of stacks because they must now also store the values of logical variables. Recall from Sect. 2.2.3 that a stack is a mapping from local variables to values. From now on, we will assume a stack $s \in \mathit{Stacks}$ to be a mapping from local variables *and* logical variables to values. That is, *Stacks* is the set

$$\prod_{z \in Var \cup LVar} dom([z]) \ .$$

This change has no consequences for the meanings of programs since logical variables are not allowed to occur in programs. Also recall from Sect. 2.2.3 that a state (s, h) is valid if h is valid and s is consistent with h. A stack s is said to be consistent with heap h if all variables (including the logical variables) either reference an object that exists in h or *null*.

Logical expressions are evaluated in two states: the current program state (s, h), and a freeze state (s', h') that assigns values to expressions of the form $\mathsf{old}(e)$. The freeze state is a copy of the initial state of a method execution. We use the following evaluation function for logical expressions.

$$\mathcal{L}[\![\_]\!] : \mathit{QExpr} \times (\mathit{Stacks} \times \mathit{Heaps}) \times (\mathit{Stacks} \times \mathit{Heaps}) \rightharpoonup \mathit{Val}$$

$$
\begin{aligned}
\mathcal{L}[\![\mathsf{null}]\!](s,h)(s',h') &= \mathit{null}\\
\mathcal{L}[\![\mathsf{this}]\!](s,h)(s',h') &= s(\mathsf{this})\\
\mathcal{L}[\![u]\!](s,h)(s',h') &= s(u)\\
\mathcal{L}[\![q.x]\!](s,h)(s',h') &= \begin{cases} \bot & \text{if } v \in \{\mathit{null}, \bot\}\\ h(v)(\mathsf{origin}([q],x))(x) & \text{otherwise}\end{cases}\\
\mathcal{L}[\![\ (C)q\ ]\!](s,h)(s',h') &= \begin{cases} v & \text{if } v \in \mathit{dom}(C)\\ \bot & \text{otherwise}\end{cases}\\
\mathcal{L}[\![q \text{ instanceof } C]\!](s,h)(s',h') &= \begin{cases} \bot & \text{if } v = \bot\\ \mathit{tt} & \text{if } v \in \mathit{dom}(C)\\ \mathit{ff} & \text{otherwise}\end{cases}\\
\mathcal{L}[\![q_1\ ?\ q_2 : q_3]\!](s,h)(s',h') &= \begin{cases} \bot & \text{if } v_1 = \bot\\ v_2 & \text{if } v_1 = \mathit{tt}\\ v_3 & \text{if } v_1 = \mathit{ff}\end{cases}\\
\mathcal{L}[\![q_1 = q_2]\!](s,h)(s',h') &= \begin{cases} \bot & \text{if } v_1 = \bot \text{ or } v_2 = \bot\\ \mathit{tt} & \text{if } v_1 = v_2 \neq \bot\\ \mathit{ff} & \text{otherwise}\end{cases}\\
\mathcal{L}[\![\mathsf{op}(q_1,\ldots,q_n)]\!](s,h)(s',h') &= \begin{cases} \bot & \text{if } v_i = \bot\\ \overline{\mathsf{op}}(v_1,\ldots,v_n) & \text{otherwise}\end{cases}\\
\mathcal{L}[\![z]\!](s,h)(s',h') &= s(z)\\
\mathcal{L}[\![\mathsf{old}(e)]\!](s,h)(s',h') &= \mathcal{E}[\![e]\!](s',h')\\
\mathcal{L}[\![z[q]]\!](s,h)(s',h') &= \begin{cases} f(v) & \text{if } 0 \leq v < n, \text{where } s(z) = (f,n)\\ \bot & \text{otherwise}\end{cases}\\
\mathcal{L}[\![\mathsf{length}(z)]\!](s,h)(s',h') &= n, \text{ where } s(z) = (f,n)\\
\mathcal{L}[\![\mathsf{undefined}]\!](s,h)(s',h') &= \bot\\
\mathcal{L}[\![\mathsf{defined}(q)]\!](s,h)(s',h') &= \begin{cases} \mathit{tt} & \text{if } v \neq \bot\\ \mathit{ff} & \text{otherwise}\end{cases}
\end{aligned}
$$

Figure 3.2: Evaluation of logical expressions. In the left hand sides we assume that v and v_i, with $i \in \{1,2,\ldots\}$, abbreviate $\mathcal{L}[\![q]\!](s,h)(s',h')$ and $\mathcal{L}[\![q_i]\!](s,h)(s',h')$, respectively.

Its second argument is the current state, the third argument is the freeze state. We say that a freeze state (s',h') is compatible with a current state (s,h) if $\mathit{dom}(h') \subseteq \mathit{dom}(h)$. The evaluation function is undefined if the freeze state is incompatible with the current state. Thus evaluation of $\mathsf{old}(e)$ cannot result in a value that does not exist in (s,h).

The definition of $\mathcal{L}[\![q]\!](s,h)(s',h')$ can be found in Fig. 3.2. The evaluation of logical expressions follows the evaluation of program expressions. The new cases are rather straightforward. Note that only $\mathcal{L}[\![\mathsf{old}(e)]\!](s,h)(s',h')$ depends on the freeze state.

Logical expression evaluation is type-safe.

Lemma 3.1. *For every expression q, every state (s,h), and every compatible*

freeze state (s', h') *we have*

$$\mathcal{L}[\![q]\!](s,h)(s',h') \in dom([q]) \cup \{\bot\} \ .$$

Proof. By structural induction on q. □

The following lemma states that program expressions have the same values in assertions as in programs.

Lemma 3.2. *For every program expression* e*, every state* (s, h)*, and every compatible freeze state* (s', h') *we have* $\mathcal{E}[\![e]\!](s,h) = \mathcal{L}[\![e]\!](s,h)(s',h')$*.*

Proof. By structural induction on e. □

We use a different evaluation function for precondition expressions.

$$\mathcal{N}[\![\_]\!] : PExpr \times (Stacks \times Heaps) \to Val$$

This evaluation function has one argument less because the values of precondition expressions do not depend on a freeze state. The value of $\mathcal{N}[\![p]\!](s,h)$ is also defined by induction on the structure of the precondition expression p; its definition coincides with the definition of $\mathcal{L}[\![p]\!](s,h)(s,h)$ in all cases, and is therefore omitted.

We have for $\mathcal{N}[\![\_]\!]$ the following counterpart of Lemma 3.2.

Lemma 3.3. *For every program expression* e *and every state* (s, h) *we have* $\mathcal{E}[\![e]\!](s,h) = \mathcal{N}[\![e]\!](s,h)$*.*

Proof. By structural induction on e. □

Formulas from the assertion language are also evaluated in two states. We use the following evaluation function to evaluate formulas.

$$\mathcal{A}[\![\_]\!] : QForm \times (Stacks \times Heaps) \times (Stacks \times Heaps) \rightharpoonup \{tt, ff\}$$

The range of the evaluation function reveals that formulas always evaluate to tt or ff, which implies that we are dealing with a classical two-valued logic. This should come as a surprise, since the formulas are built from expressions whose values may be undefined. One would expect that the values of formulas could also be undefined. However, a three-valued logic would complicate many definitions. Therefore we choose to assign the value ff to expressions whose value is undefined by means of the following definition of $\mathcal{A}[\![q]\!](s,h)(s',h')$.

$$\mathcal{A}[\![q]\!](s,h)(s',h') = \begin{cases} tt & \text{if } \mathcal{L}[\![q]\!](s,h)(s',h') = tt \\ ff & \text{otherwise} \end{cases}$$

The evaluation of formulas that are composed by means of the classical propositional connectives is standard.

$$\begin{aligned} \mathcal{A}[\![\neg Q]\!](s,h)(s',h') &= \begin{cases} tt & \text{if } \mathcal{A}[\![Q]\!](s,h)(s',h') = ff \\ ff & \text{otherwise} \end{cases} \\ \mathcal{A}[\![Q_1 \wedge Q_2]\!](s,h)(s',h') &= \begin{cases} tt & \text{if } \mathcal{A}[\![Q_1]\!](s,h)(s',h') = tt \\ & \text{and } \mathcal{A}[\![Q_2]\!](s,h)(s',h') = tt \\ ff & \text{otherwise} \end{cases} \end{aligned}$$

We would like to point out that these definitions imply that he validity of an assertion $\neg p$ does not imply that the value of p is ff; its value can also be undefined. However, one can use the formula $p = \mathsf{false}$ instead of $\neg p$ to test whether the value of p is really ff. By contrast, a formula p is always equivalent to $p = \mathsf{true}$.

The quantification domain of primitive types and sequences of values of a primitive type does not depend on the state.

$$\mathcal{A}[\![(\exists z : t \bullet Q)]\!](s,h)(s',h') = \begin{cases} \mathit{tt} & \text{if there exists a } v \in \mathit{dom}(t) \text{ such} \\ & \text{that } \mathcal{A}[\![Q]\!](s[z \mapsto v],h)(s',h') = \mathit{tt} \\ \mathit{ff} & \text{otherwise, for } t \in \{\mathsf{boolean}(^*), \mathsf{int}(^*)\} \end{cases}$$

Quantification over objects of a particular class is always restricted to the set of objects of that class that exist in the current state. The following definition, where $V = \mathit{dom}(C) \cap \mathit{dom}(h)$, defines the meaning of quantification of a the objects of a particular class.

$$\mathcal{A}[\![(\exists z : C \bullet Q)]\!](s,h)(s',h') = \begin{cases} \mathit{tt} & \text{if there exists a } v \in V \text{ such} \\ & \text{that } \mathcal{A}[\![Q]\!](s[z \mapsto v],h)(s',h') = \mathit{tt} \\ \mathit{ff} & \text{otherwise} \end{cases}$$

Finally, we also define the meaning of quantification over sequences of objects. Let $V = \{\bar{v} \mid \bar{v} \in \mathit{dom}(C^*) \text{ and } \bar{v} \sqsubseteq \mathit{dom}(h)\}$. Then

$$\mathcal{A}[\![(\exists z : C^* \bullet Q)]\!](s,h)(s',h') = \begin{cases} \mathit{tt} & \text{if there exists a sequence } \bar{v} \in V \text{ such} \\ & \text{that } \mathcal{A}[\![Q]\!](s[z \mapsto \bar{v}],h)(s',h') = \mathit{tt} \\ \mathit{ff} & \text{otherwise .} \end{cases}$$

We will write $(s,h)(s',h') \models Q$ as shorthand for $\mathcal{A}[\![Q]\!](s,h)(s',h') = \mathit{tt}$. An assertion Q is valid, which we express by writing $\models Q$, if $(s,h)(s',h') \models Q$ for every state (s,h), and every compatible state (s',h').

Preconditions are evaluated in a single state. We use the following evaluation function for preconditions.

$$\mathcal{P}[\![\_]\!] : \mathit{PForm} \times (\mathit{Stacks} \times \mathit{Heaps}) \to \{\mathit{tt}, \mathit{ff}\}$$

We have

$$\mathcal{P}[\![p]\!](s,h) = \begin{cases} \mathit{tt} & \text{if } \mathcal{N}[\![p]\!](s,h) = \mathit{tt} \\ \mathit{ff} & \text{otherwise .} \end{cases}$$

The definition of $\mathcal{P}[\![P]\!](s,h)$ is similar to the definition of $\mathcal{A}[\![P]\!](s,h)(s,h)$ in all other cases.

The following lemma shows that the value of a precondition is independent of the freeze state.

Lemma 3.4. *For every precondition expression p, every precondition P, every current state (s,h), and every compatible freeze state (s',h') we have*

1. $\mathcal{N}[\![p]\!](s,h) = \mathcal{L}[\![p]\!](s,h)(s',h')$, *and*

2. $\mathcal{P}[\![P]\!](s,h) = \mathcal{A}[\![P]\!](s,h)(s',h')$

Proof. By structural induction on p and P. Note that $Pexpr \subseteq Qexpr$, and that $PForm \subseteq QForm$. The proof of the second claim depends on the first claim. □

For preconditions, we write $(s,h) \models P$ as abbreviation of $\mathcal{P}[\![P]\!](s,h) = tt$. A precondition P is valid, denoted by $\models P$, if $(s,h) \models P$ for every state (s,h).

Remark 3.1. *We will often simply write* $(s,h) \models Q$ *instead of* $(s,h)(s',h') \models Q$ *if it is evident that* $Q \in PForm$*. Lemma 3.4 justifies this shorter form by showing that the validity of* $(s,h)(s',h') \models Q$ *does not depend on the freeze state* (s',h') *if* $Q \in PForm$*.*

In the following sections, we will need a syntactical operation $[./\mathsf{old}(.)]$ that replaces all expressions of the form $\mathsf{old}(e)$ in an assertion by e. Thus it translates each formula into a precondition, and each logical expression into a precondition expression. It is defined by induction on the structure of logical expressions and assertions. Its characteristic case is as follows.

$$\mathsf{old}(e)[./\mathsf{old}(.)] \equiv e$$

Its definition corresponds to the usual definition of structural substitution for all other cases. Clearly, $q[./\mathsf{old}(.)] \in PExpr$ for every $q \in QExpr$, and similarly $Q[./\mathsf{old}(.)] \in PForm$ for every $Q \in QForm$.

The following lemma describes the effect of the operation in terms of the semantics of assertions.

Lemma 3.5. *For every assertion* Q*, and every state* (s,h) *we have*

$$\mathcal{P}[\![Q[./\mathsf{old}(.)]]\!](s,h) = \mathcal{A}[\![Q]\!](s,h)(s,h) \ .$$

Proof. The proof proceeds by structural induction on Q. The base case requires us to prove by structural induction on q that

$$\mathcal{L}[\![q[./\mathsf{old}(.)]]\!](s,h)(s,h) = \mathcal{L}[\![q]\!](s,h)(s,h)$$

for every expression q and every state (s,h). The only interesting case is computed as follows.

$$\begin{array}{ll} & \mathcal{L}[\![\mathsf{old}(e)[./\mathsf{old}(.)]]\!](s,h)(s,h) \\ = & \mathcal{L}[\![e]\!](s,h)(s,h) & \{ \text{ def. } [./\mathsf{old}(.)] \ \} \\ = & \mathcal{E}[\![e]\!](s,h) & \{ \text{ Lemma 3.2 } \} \\ = & \mathcal{L}[\![\mathsf{old}(e)]\!](s,h)(s,h) & \{ \text{ def. } \mathcal{L}[\![\_]\!] \ \} \end{array}$$

□

3.2 Hoare Triples

There is a strong connection between proof outlines logics [Flo67, OG76] and Hoare logics [Hoa69]. A proof outline can be viewed as a convenient representation of a proof in a Hoare logic [OG76]. Hoare logics are based on Hoare triples, which are a means to specify the input-output behavior of statements. One of the aims of this thesis is to elucidate the correspondence between proof outlines logics and Hoare logics. For this purpose we will define the validity of proof outlines, which are introduced in the following two sections, in terms of the validity of Hoare triples. Hoare triples are studied in this section.

We distinguish two types of Hoare triples. The first type specifies the behavior of statements, and has the form $\{Q\}S\{Q'\}$. The second type of Hoare triples specify the behavior of methods. A Hoare triple of this type has the form $\{P\}m@C\{Q\}$. It specifies the precondition P and the postcondition Q of method m in class C.

Informally, a Hoare triple $\{Q\}S\{Q'\}$ means that every terminating computation of S that starts in a state that satisfies Q, terminates in a state that satisfies Q'. This interpretation of Hoare triples is known as *partial correctness*. The following definition of the validity of a Hoare triple formalizes this notion.

Definition 3.2. *A Hoare triple of the form $\{Q\}S\{Q'\}$ is valid, which is expressed by $\models \{Q\}S\{Q'\}$, if and only if for every current state (s,h), every compatible freeze state (s',h'), and every computation $\langle S,(s,h)\rangle \to (s'',h'')$,*

$$(s,h)(s',h') \models Q \text{ implies } (s'',h'')(s',h') \models Q' \ .$$

A Hoare triple of the form $\{P\}m@C\{Q\}$ describes the behavior of the implementation of method m declared in class C. It corresponds to the interface specification of a method in an annotated program, which is the specification of a method that is visible to its clients. We use a special-purpose logical variable result to denote the result value in the postcondition Q of a method. Its type $[\mathsf{result}]$ is the return type of the method.

The validity of this second type of Hoare triple is defined as follows.

Definition 3.3. *Let S be the body of method m in class C. Let e be the expression that denotes its return value. Then the Hoare triple $\{P\}m@C\{Q\}$ is valid, which is denoted by $\models \{P\}m@C\{Q\}$, if and only if for every current state (s,h) and every computation $\langle S,(s,h)\rangle \to (s',h')$ such that $(s,h) \models P$ and $\mathcal{E}[\![e]\!](s',h') = v \neq \bot$ we have that $(s'[\mathsf{result} \mapsto v],h')(s,h) \models Q$.*

The state modification $[\mathsf{result} \mapsto \mathcal{E}[\![e]\!](s',h')]$ in the definition models the effect of a (virtual) assignment $\mathsf{result} := e$ in state (s',h'). Naturally, it should be omitted if method m has return type void.

The effect of the above assignment on the validity of a formula Q can be neutralized by a syntactical operation $[e/\mathsf{result}]$. This operation corresponds to the structural substitution of result by e in a formula Q. However, its most

important case is slightly more complex.

$$z[e/\mathsf{result}] \equiv \begin{cases} e & \text{if } z \equiv \mathsf{result} \text{ and } [\mathsf{result}] = [e] \\ ([\mathsf{result}])e & \text{if } z \equiv \mathsf{result} \text{ and } [e] \prec [\mathsf{result}] \\ z & \text{if } z \not\equiv \mathsf{result} \end{cases}$$

Recall from Sect. 2.2.2 that $[e] \prec [\mathsf{result}]$ expresses that the type of e is a proper subtype of the type of result. The operation $[e/\mathsf{result}]$ is undefined if $[e] \not\preceq [\mathsf{result}]$. However, the typing rule for expressions that denote the return value stipulates that the type of e must always be a subtype of the return type of the method, which is also the type of result. (see Sect. 2.2.2). The given definition of structural substitution is more complex than usual because it is essential that the operation preserves the type of the expression. We motivate this requirement in Chapter 4.

The following lemma states that an assertion $Q[e/\mathsf{result}]$ has the same value in the initial state as the assertion Q after the assignment $\mathsf{result} := e$.

Lemma 3.6. *For every assertion Q, every state (s, h), and expression e such that $[e] \preceq [\mathsf{result}]$ and $\mathcal{E}[\![e]\!](s, h) \neq \bot$, we have*

$$\mathcal{A}[\![Q[e/\mathsf{result}]]\!](s, h)(s', h') = \mathcal{A}[\![Q]\!](s[\mathsf{result} \mapsto \mathcal{E}[\![e]\!](s, h)], h)(s', h') \ .$$

The lemma is a specific instance of Lemma 4.3 in Chapter 4.

We use the substitution operation $[e/\mathsf{result}]$ in the following lemma, which describes when the specification of a method follows from the specification of its body.

Lemma 3.7. *Let S be the body of method m in class C. Let e be the expression that denotes its return value. Let $\models \{Q'\}S\{Q''\}$. Then*

$$\models P \rightarrow Q'[./\mathsf{old}(.)] \text{ and } \models Q'' \rightarrow Q[e/\mathsf{result}] \text{ imply } \models \{P\}m@C\{Q\} \ .$$

Proof. Let $(s, h) \models P$, and let $\langle S, (s, h)\rangle \rightarrow (s', h')$ be a computation such that $\mathcal{E}[\![e]\!](s', h') \neq \bot$. According to Def. 3.3 we must show that

$$(s'[\mathsf{result} \mapsto \mathcal{E}[\![e]\!](s', h')], h')(s, h) \models Q \ .$$

From $(s, h) \models P$ and the first implication follows $(s, h)(s, h) \models Q'[./\mathsf{old}(.)]$. The latter formula is equivalent to $(s, h)(s, h) \models Q'$ according to Lemma 3.5. We can then use Def. 3.2 and $\models \{Q'\}S\{Q''\}$ to prove that $(s', h')(s, h) \models Q''$. By the second implication this latter formula implies $(s', h')(s, h) \models Q[e/\mathsf{result}]$. The desired goal then follows from Lemma 3.6 and the previous formula. □

3.3 Annotated Programs

This section describes the syntax of *annotated programs*. We will annotate method bodies with assertions that describe the state at all control points.

Moreover, an interface specification consisting of a precondition and a postcondition will be assigned to each method declaration. A precondition describes the states in which clients have the right to call the method. A postcondition describes the states in which the execution of the method may terminate provided that the execution was initiated by a client in a state that satisfies the precondition. Thus they form a contract between the implementor of the method and its clients [Mey88].

An annotated program is a COORE program in which each method is annotated. Annotated methods are assumed to have the following form.

$$\begin{array}{l}\mathsf{requires}\ P;\\ \mathsf{ensures}\ Q;\\ t\ m(\bar{p})\ \{\ \mathsf{assert}\ Q;\ S\ ;\ \mathsf{assert}\ Q;\ \mathsf{return}\ e\ \}\end{array}$$

The requires keyword precedes the precondition of the method; the ensures keyword indicates its postcondition. We require that the postcondition does not contain free occurrences of local variables other than the local variables that occur in expressions of the form $\mathsf{old}(e)$. Moreover, the set of local variables in such expressions must be subsumed by the parameter set $\bar{p}$.

The special-purpose variable result will denote the result value in the postcondition. It is not allowed to occur in any other assertion.

The body S of the method is enclosed by two assertions. Each assertion is preceded by the keyword assert. However, we will also need annotation inside certain statements. We therefore provide a grammar for *annotated* statements below.

$$\begin{array}{rcl} S \in \mathit{Stat}' & ::= & t\ u \mid u := e \mid e.x := e \mid S\ ;\ \mathsf{assert}\ Q;\ S \\ & \mid & u := \mathsf{new}\ C(\bar{e}) \mid e.m(\bar{e}) \mid u := e.m(\bar{e}) \\ & \mid & \mathsf{if}\ (e)\ S\ \mathsf{else}\ S \mid \mathsf{while}\ (e)\ \mathsf{assert}\ Q;\ S \end{array}$$

Note that we use S as a typical element of both the set of statements *Stat* and the set of annotated statements *Stat′*. We do not expect that this will cause confusion.

It is not difficult to see that this annotation strategy ensures that every semicolon is followed by an assertion. The intermediate assertion in an annotated statement of the form

$$\mathsf{assert}\ Q;\ S_1\ ;\ \mathsf{assert}\ Q;\ S_2\ ;\ \mathsf{assert}\ Q;$$

describes the states that may result from executing S_1 in a state that satisfies the precondition of the composed statement. The additional assertion in a while statement is called its invariant.

Example 3.2. *The annotated class in Fig. 3.3 illustrates our specification strategy. The method getAndSetAge assigns the value of its parameter to the age field of the receiver, and returns the previous value of the field.*

```
class Person {
  int age;

  requires true;
  ensures this.age = old(age) ∧ oldAge = old(this.age);
  int  getAndSetAge(int age) {
    assert age = old(age) ∧ this.age = old(this.age);
    int oldAge;
    assert age = old(age) ∧ this.age = old(this.age);
    oldAge := this.age;
    assert age = old(age) ∧ oldAge = old(this.age);
    this.age := age;
    assert this.age = old(age) ∧ oldAge = old(this.age);
    return oldAge
  }
}
```

Figure 3.3: A simple class *Person* with an annotated method.

We need to stress that the introduced assertions are assumed to have no impact at all on the execution of a method or statement. We expect the compiler to ignore all introduced annotations. In particular, we do not assume that the program checks at runtime if the assertion that is assigned to its current control point holds. Java supports this type of checking, which is usually called *runtime assertion checking*, since version 1.4. It is also available in C# [Mok03], and some researchers are trying to enrich its checking facilities [BLS05]. Cheon and Leavens describe how Java assertions written in JML can be checked [CL02]. Instead, we will develop a proof system that enables us to prove statically that no assertion will ever be violated if the program annotations are correct.

Not every annotated method constitutes a proof for the validity of its interface specification. The following section investigates the question which annotated methods are valid proof outlines for the interface specifications of their methods.

3.4 Proof Outlines

In the previous section we introduced an annotation strategy for methods. The resulting annotated methods are supposed to justify the interface specifications of the methods. We will first formally define a notion of validity for method interface specifications.

Definition 3.4. *The interface specification of an annotated method of the form*

$$\mathsf{requires}\ P;\ \mathsf{ensures}\ Q;\ (t \mid \mathsf{void})\ m(\bar{p})\ \{\ \ldots\}$$

S	$PO(\mathsf{assert}\ Q;\ S\ ;\ \mathsf{assert}\ Q';)$
$t\ u$	$\{\{Q\}\ t\ u\ \{Q'\}\}$
$u := e$	$\{\{Q\}\ u := e\ \{Q'\}\}$
$e.x := e'$	$\{\{Q\}\ e.x := e'\ \{Q'\}\}$
$S_1\ ;\ \mathsf{assert}\ Q'';\ S_2$	$PO(\mathsf{assert}\ Q;\ S_1;\ \mathsf{assert}\ Q'';)$ $\cup\ PO(\mathsf{assert}\ Q'';\ S_2;\ \mathsf{assert}\ Q';)$
$u := \mathsf{new}\ C(\bar{e})$	$\{\{Q\}\ u := \mathsf{new}\ C(\bar{e})\ \{Q'\}\}$
$e.m(\bar{e})$	$\{\{Q\}\ e.m(\bar{e})\ \{Q'\}\}$
$u := e.m(\bar{e})$	$\{\{Q\}\ u := e.m(\bar{e})\ \{Q'\}\}$
$\mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2$	$PO(\mathsf{assert}\ Q \wedge e;\ S_1;\ \mathsf{assert}\ Q';) \cup$ $PO(\mathsf{assert}\ Q \wedge e = \mathsf{false};\ S_2;\ \mathsf{assert}\ Q';)$
$\mathsf{while}\ (e)\ \mathsf{assert}\ Q'';\ S$	$PO(\mathsf{assert}\ Q'' \wedge e;\ S;\ \mathsf{assert}\ Q'';)$ $\cup\ \{Q \to Q'', Q'' \wedge e = \mathsf{false} \to Q'\}$

Table 3.1: The set of proof obligations of annotated statements.

in class C is valid if and only if $\models \{P\}\ m@C\ \{Q\}$.

Note that we have defined the validity of an interface specification in terms of the validity of a Hoare triple. We will also use this strategy in the definition of the validity of method annotations. A method annotation will be said to be valid if its set of proof obligations are valid. This set of proof obligations contains both formulas from the assertion language and Hoare triples.

We first define the set of proof obligations for an annotated method body $\mathsf{assert}\ Q;\ S\ \mathsf{assert}\ Q'$; by induction on the structure of S. The function PO yields the set of proof obligations. Its definition can be found in Table 3.1.

For all basic annotated statements validity simply means validity of the corresponding Hoare triple. The intermediate assertions in annotated compound statements are used to define the validity of the annotation in terms of the validity of the annotation of their parts.

The following theorem justifies the above validity definition.

Theorem 3.8. *Let S be an arbitrary annotated statement, and let Q, Q' be arbitrary assertions. If for all $\phi \in PO(\mathsf{assert}\ Q;\ S;\ \mathsf{assert}\ Q';)$ we have $\models \phi$ then $\models \{Q\}\ S\ \{Q'\}$.*

Proof. The proof is by induction on the structure of S. The theorem holds trivially for basic statements. The three cases that involve compound statements are covered by Lemma 3.9, Lemma 3.10, and Lemma 3.11 below. □

Lemma 3.9. *Let $\models \{Q\}\ S_1\ \{Q''\}$ and $\models \{Q''\}\ S_2\ \{Q'\}$ for arbitrary statements S_1 and S_2 and arbitrary assertions Q, Q', and Q''. Then $\models \{Q\}\ S_1;\ S_2\ \{Q'\}$.*

Lemma 3.10. *Let $\models \{Q \wedge e\}\ S_1\ \{Q'\}$ and $\models \{Q \wedge e = \mathsf{false}\}\ S_2\ \{Q'\}$ for arbitrary statements S_1 and S_2, and arbitrary assertions Q and Q'. Then $\models \{Q\}\ \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2\ \{Q'\}$.*

Lemma 3.11. *Let* $\models \{Q'' \wedge e\}\ S\ \{Q''\}$, $\models Q \rightarrow Q''$ *and* $\models Q'' \wedge e = \mathsf{false} \rightarrow Q'$ *for an arbitrary statement* S*, and arbitrary assertions* Q*,* Q' *and* Q''*. Then* $\models \{Q\}\ \mathsf{while}\ (e)\ S\ \{Q'\}$*.*

The last lemma is the most difficult one to prove. We give its proof below.

Proof. We first prove by natural induction on n that for every state (s, h) and every compatible freeze state (s', h') such that $(s, h)(s', h') \models Q''$, it is the case that

$$\langle \mathsf{while}\ (e)\ S, (s, h) \xrightarrow{n} (s'', h'')\rangle \text{ implies } (s'', h'')(s', h') \models Q' \ . \qquad (3.1)$$

Let $n = 0$. The assumption of Rule Wh_1 and Lemma 3.2 together imply that $(s, h)(s', h') \models e = \mathsf{false}$. We have $(s, h)(s', h') \models Q'' \wedge e = \mathsf{false} \rightarrow Q'$. By modus ponens then $(s, h)(s', h') \models Q'$. Rule Wh_1 also reveals that $(s, h) = (s'', h'')$, and therefore $(s'', h'')(s', h') \models Q'$.

Now assume that $\langle \mathsf{while}\ (e)\ S, (s, h) \xrightarrow{n+1} (s'', h'')$ and $(s, h)(s', h') \models Q''$. According to Rule Wh_2, there must be an intermediate state (s_1, h_1) such that $\mathcal{E}[\![e]\!](s, h) = tt$, $\langle S, (s, h)\rangle \rightarrow (s_1, h_1)$, and $\langle \mathsf{while}\ (e)\ S, (s_1, h_1)\rangle \xrightarrow{n} (s'', h'')$. The first of these three assumptions and Lemma 3.2 imply that $(s, h)(s', h') \models e$. By $\{Q'' \wedge e\}\ S\ \{Q''\}$ and Def. 3.2 we may then conclude that $(s_1, h_1)(s', h') \models Q''$. Then (3.1) follows from the induction hypothesis.

We now return to the main lemma. Let $\langle \mathsf{while}\ (e)\ S, (s, h)\rangle \rightarrow (s'', h'')$ and $(s, h)(s', h') \models Q$. By modus ponens we obtain $(s, h)(s', h') \models Q''$ from the last assumption and $\models Q' \rightarrow Q''$. By Rule Wh we know that there must be some $n \in \{0, 1, \ldots\}$ such that $\langle \mathsf{while}\ (e)\ S, (s, h)\rangle \xrightarrow{n} (s'', h'')$. Then (3.1) says that $(s'', h'')(s', h') \models Q'$. □

After having defined the proof obligations of annotated method bodies, we can now also define the set of proof obligations of an entire method, which includes the proof obligations of its body.

Definition 3.5. *The set of proof obligations of an annotated method of the form*

$$\begin{array}{l}\mathsf{requires}\ P;\\ \mathsf{ensures}\ Q;\\ t\ m(\bar{p})\ \{\ \ \mathsf{assert}\ Q';\ S;\ \mathsf{assert}\ Q'';\ \mathsf{return}\ e'\ \ \}\end{array}$$

is obtained by adding the implications $P \rightarrow Q'[./\mathsf{old}(.)]$ *and* $Q'' \rightarrow Q[e'/\mathsf{result}]$ *to the set* $PO(\mathsf{assert}\ Q';\ S;\ \mathsf{assert}\ Q'';)$*.*

Naturally, we wish to know whether the proof obligations ensure that the interface specification of a method is valid, as defined by Def. 3.4. The following theorem ensures us that this is the case.

Theorem 3.12. *Let method* m *in class* C *be annotated as in Def. 3.5. The interface specification of this method is valid if its set of proof obligations are valid.*

Proof. The result follows from Lemma 3.7 and Theorem 3.8. □

Let us summarize the series of definitions and results that we have introduced in this chapter in order to discuss some of its implications. We have given a formal syntax of annotated programs, and we have shown that the validity of the annotation boils down to the validity of a set of formulas from the assertion language, and a set of Hoare triples (Theorem 3.12). The set of proof obligations has been defined inductively, and it can be computed automatically for every annotated program.

This positive result raises the question if there is a way to even further simplify the set of proof obligations. In particular, it raises the question if there is a means to get rid of the obligation to reason about Hoare triples. The next three chapters will present techniques that enable us to reduce such proof obligations to the validity of formulas from the assertion language. The availability of such techniques is the essential difference between a Hoare logic (for reasoning about the validity of Hoare triples) and a proof outline logic.

3.5 Related Specification Languages

Early formalisms for program verification [Flo67, Hoa69, Apt81] are all based on multi-sorted first-order predicate logic. It is, however, not clear if first-order logic is also sufficiently expressive for reasoning about object-oriented programs with dynamically allocated objects. Most early work on Hoare logics targets languages which have states that consist of a fixed set of variables.

We follow the approach advocated by America and De Boer [dB91, AdB94, dB99, dB02] that extends the domain of the logic with finite sequences. We show in this thesis that this extension is sufficient to obtain a complete proof system for object-oriented languages with subtype polymorphism and inheritance provided that the language contains a construct to access hidden variables, and a means to test if an object belongs to a particular domain (cf. Chapter 7). We added casts and the instanceof operator to the language for these reasons.

Another way to strengthen the expressiveness of the language is to add an explicit heap reference to the language as is done in the work of Poetzsch-Heffter and Müller [PH97, PHM99, Mül02]. Their specifications refer to the current heap (or store) by means of the `$` constant, and encode heap modifications as operators on the heap. For example, $\$(L)$ would denote the value of location L in the current heap, and $\$\langle L := \mathit{int}(5)\rangle$ denotes the heap that results from updating location L with integer value 5 (cf. [Mül02]). The meaning of the heap operators is specified using a series of additional axioms. Existence of objects is encoded using an auxiliary *alive* field. Our specification language is more closely tailored to the abstraction level of the programming language. For example, objects that do not exist neither play a role in the programming language nor in our assertion language.

The Object Constraint Language (OCL, [WK98]) is a subset of the Unified Modeling Language (UML) that can be used to add invariants and pre- and postconditions to UML diagrams. UML, and consequently OCL, is is not

tailored to a specific programming language.

Object-Z [Smi92, Smi00] is an extension of the Z [Spi92] specification language. It brings operation schemas and state schemas together in class definitions. The state schemas may include invariants, and operation schemas describe pre- and postconditions of operations using the primed variable notation that is typical for Z. However, these formulas can only restrict the values of the attributes of the object to which the operation is applied. Thus the standard Object-Z annotations are much weaker than our specifications. This weakness is compensated in some publications on Object-Z by the addition of temporal logic formulas (see, e.g., [Smi92]).

Several specification languages have been designed to specify programs written in a particular language. Perhaps the most notable example is the Java Modeling Language (JML), which is a behavioral interface specification language for Java programs [LBR04, LCC+02]. It is a rich language that supports among others pre- and postconditions, frame axioms (using assignable clauses), invariants, and constraints. JML formulas are built from side effect free (pure) Java expressions. The specifications may use types and methods from a library of specification classes. More abstract specifications can be written using model fields [BP03].

JML has grown so rapidly that most tools that claim to support the language in fact only support a subset of JML [JMR04, CK05]. The language is also unavoidably subject to constant change, and it is not backed by a rigorous formal semantics. Most theoretical work on specification and verification of object-oriented languages (including this thesis) is based on JML-like languages of a more manageable size.

The Spec# programming language [BNSS04] extends the .NET programming language C# with specification constructs like pre- and postconditions, non-null types and invariants. Its invariant methodology differs from that of JML (cf. Chapter 9). It also forces C# programmers to list all checked exceptions in the interface specification of a method [LS04].

Both JML and Spec# allow calls to pure methods in formulas. Cok [Cok04] describes how method calls can be translated into the logic of ESC/Java2. Barnett et al. [BNSS04] and Naumann [Nau05] discuss observational purity, a criterion under which methods that are not entirely pure can be used in method specifications. The embedding of method calls in the specification language is considered outside the scope of this thesis.

We ignore object deallocation due to garbage collection in the semantics of the programming language, and consequently also in the specification language. This seems a natural decision because garbage collection is supposed to have no effect on the execution of a program. However, the final state of a program may differ if garbage collection takes place because unreachable objects may have been removed. Calcagno et al. [COB03] studied the consequences of garbage collection (object deallocation) on program specifications. They essentially propose to alter the semantics of quantification in such a way that non-existing objects are also included.

Chapter 4

Reasoning about Assignments

This chapter shows how one can compute the verification conditions that check the validity of Hoare triples of basic assignments. More in particular, it describes the verification conditions of Hoare triples of the following three forms.

$$\{Q\}\ t\ u\ \{Q'\} \quad \{Q\}\ u := e\ \{Q'\} \quad \{Q\}\ e.x := e'\ \{Q'\}$$

The Hoare triples of the first kind, which specify local variable declarations, do not contain an assignment symbol. However, a local variable declaration $t\ u$ assigns the default value of type t to the local variable u. In other words, it is equivalent to the assignment $u := \mathsf{def}(t)$, where $\mathsf{def}(t)$ denotes an expression whose value is the default value of t in every state.

There are two features in object-oriented programming that complicate reasoning about assignments: field shadowing and aliasing. We will explain the impact of these features in the following two sections. The following section discusses field shadowing, and it presents a weakest precondition calculus for assignments to local variables. The weakest precondition calculus can be used to calculate the verification condition of Hoare triples of local assignments. Section 4.2 discusses aliasing in the context of a weakest precondition calculus for field assignments. Section 4.3 presents the strongest postconditions of assignments. We finish this chapter with some historical notes and a discussion of related work.

4.1 Local Assignments and Field Shadowing

Hoare's original axiom for assignments has the form $\{Q^x_e\}\ x := e\ \{Q\}$, where Q^x_e denotes the assertion that is obtained by replacing all free occurrences of x by e in Q [Hoa69]. The assertion Q^x_e is the *weakest precondition* of $x := e$ with respect to Q. The weakest precondition is valid in those states for which holds

that each terminating execution of $x := e$ started in that state terminates in a state that satisfies Q.

Hoare's axiom is appropriate for simple imperative languages. But is it also valid in object-oriented programs? We will show in this section that the axiom does not hold for assignments in object-oriented languages with subtype polymorphism and field shadowing.

The predicate Q_e^x is the weakest precondition of an assignment $x := e$ because the variable x has the original value of e after the assignment. However, in languages with subtype polymorphism, it is in general not the case that the types of x and e are equal. We explained in Section 2.2.2 that such languages only require that the type of e is a subtype of the type of x. This typing rule, in combination with field shadowing, invalidates Hoare assignment axiom, as is shown by the following example.

Example 4.1. *Consider a class Citizen with a field num* : int *that represents a citizen's social security number. Suppose that someone writes a subclass Student of class Citizen in which he/she declares another field num* : int*, which is used to store a student number. Each Student-object now has two fields with the identifier num since class Student also inherits the fields declared in its superclass Citizen.*

Now consider the following Hoare triple.

$$\{std.num = 100\}\ ctz := std\ \{ctz.num = 100\}$$

We assume here that std is a local variable of type Student, and that ctz is a local variable of type Citizen. The assignment ctz := std is valid because $[std] \preceq [ctz]$.

Note that the Hoare triple is a valid instance of Hoare's assignment axiom. But the Hoare triple itself is clearly not valid! The precondition ensures that the student number of the object referenced by std is 100*, whereas the postcondition claims that the the social security of the same object is* 100*. Recall that the static type of the expression that denotes the object decides which field is meant in expressions of the form e.x. The Hoare triple would have been valid if the precondition had been* $((Citizen)std).num = 100$.

The cause of the problem in the example is that merely substituting a variable x in a logical expression q by some expression e does not result in an expression with the same static type as the original expression if the types of e and x differ. This may cause problems in expressions of the form $e.x$ because the static type of e determines to which field x of the object denoted by e the expressions refer. Without field shadowing the difference in type would be harmless because the identifier x would uniquely determine the field. However, the example has shown that we have to be careful with types when performing substitutions in languages with subtype polymorphism and field shadowing.

The issue can be solved by means of a type-preserving substitution operation $[e/v]$, where v is either a local variable u, a logical variable z, or the receiver

$$
\begin{array}{rcl}
\mathsf{null}[e/v] & = & \mathsf{null} \\
q.x[e/v] & = & (q[e/v]).x \\
(C)q[e/v] & = & (C)(q[e/v]) \\
q \text{ instanceof } C[e/v] & = & q[e/v] \text{ instanceof } C \\
q_1 \ ? \ q_2 : q_3[e/v] & = & (q_1[e/v]) \ ? \ (q_2[e/v]) : (q_3[e/v]) \\
q = q'[e/v] & = & (q[e/v]) = (q'[e/v]) \\
\mathsf{op}(q_1, \ldots, q_n)[e/v] & = & \mathsf{op}(q_1[e/v], \ldots, q_n[e/v]) \\
\mathsf{old}(e')[e/v] & = & \mathsf{old}(e') \\
z[q][e/v] & = & z[q[e/v]] \\
\mathsf{length}(z)[e/v] & = & \mathsf{length}(z) \\
\mathsf{undefined}[e/v] & = & \mathsf{undefined} \\
\mathsf{defined}(q)[e/v] & = & \mathsf{defined}(q[e/v]) \\
\neg Q[e/v] & = & \neg(Q[e/v]) \\
Q \wedge Q'[e/v] & = & (Q[e/v]) \wedge (Q'[e/v]) \\
(\exists z \bullet Q)[e/v] & = & \begin{cases} (\exists z \bullet Q) & \text{if } z \equiv v \\ (\exists z \bullet (Q[e/v])) & \text{if } z \not\equiv v \end{cases}
\end{array}
$$

Figure 4.1: The definition of the type-preserving structural substitution operation $[e/v]$.

this. This operation casts an inserted expression back to the type of the variable that it replaces. It is defined by induction on the structure of logical expressions and formulas. Its most interesting case is defined below, where v' also denotes either a local variable u, a logical variable z, or the receiver this.

$$
v'[e/v] = \begin{cases} v' & \text{if } v \not\equiv v' \\ e & \text{if } v \equiv v' \text{ and } [e] = [u] \\ ([v])e & \text{if } v \equiv v' \text{ and either } [e] \prec [v] \text{ or } [v] \prec [e] \end{cases}
$$

We have $v \equiv v'$ if v and v' are syntactical equal. The cast is omitted if v and e have the same type. The operation $[e/v]$ is undefined if $[e] \not\sim [v]$. The other cases of the definition of the substitution operation are listed in Fig. 4.1.

The cast that is introduced will not fail if we have that $[e] \preceq [v]$, which is what the type rules ensure for every assignment $v := e$. This is expressed by the lemma below.

Lemma 4.1. *For every logical expression q, and every reference type C such that $[q] \preceq C$, we have*

$$\mathcal{L}[\![(C)q]\!](s,h)(s',h') = \mathcal{L}[\![q]\!](s,h)(s',h')$$

for every current state (s,h), and every compatible freeze state (s',h').

Proof. It is not difficult to prove that $\perp \notin dom(C)$ for every reference type C. Hence if $\mathcal{L}[\![q]\!](s,h)(s',h') = \perp$ then $\mathcal{L}[\![(C)q]\!](s,h)(s',h') = \perp$ according to the definition of $\mathcal{L}[\![\_]\!]$. Now suppose that $\mathcal{L}[\![q]\!](s,h)(s',h') = v \neq \perp$. We then have

$v \in dom([q])$ according to Lemma 2.4. Then $v \in dom(C)$ follows from $[q] \preceq C$ and Lemma 2.1. The definition of $\mathcal{L}[\![\_]\!]$ then states $\mathcal{L}[\![(C)q]\!](s,h)(s',h') = v$. □

The following lemma states that $[e/v]$ preserves the type of an expression.

Lemma 4.2. *For every logical expression q, every variable v, and every expression e such that $[e] \preceq [u]$, we have $[q[e/v]] = [q]$.*

Proof. By structural induction on q. □

Unfortunately, we are not done yet. We must also face the possibility that an assignment $u := e$ goes 'wrong', i.e., that it will not terminate at all. This happens if the value of e is undefined in the state in which we try to assign the value of e to u. We can use the expression $\mathsf{defined}(e)$ to test whether this is the case.

We claim that $\mathsf{defined}(e) \rightarrow Q'[e/u]$ is the weakest precondition of an assignment $u := e$ with respect to an assertion Q'. An assertion Q is the weakest precondition of some statement S with respect to a postcondition Q' if it is a valid precondition, i.e., if $\models \{Q\}\ S\ \{Q'\}$, and if every other valid precondition is weaker than Q[1]. Hence we can prove our claim by showing that $\{\mathsf{defined}(e) \rightarrow Q'[e/u]\}\ u := e\ \{Q'\}$ is a valid Hoare triple, and that the validity of $\{Q\}\ u := e\ \{Q'\}$ implies $\models Q \rightarrow (\mathsf{defined}(e) \rightarrow Q'[e/u])$.

To prove our claim, we first need a lemma that implies that Q holds after an assignment $u := e$ if and only if $Q[e/u]$ holds in the initial state, provided that the value of e is defined.

Lemma 4.3. *For every assertion Q, every state (s,h), every compatible freeze state (s',h'), every variable v, and every expression e such that $[e] \preceq [v]$, we have that $\mathcal{E}[\![e]\!](s,h) \neq \bot$ implies*

$$\mathcal{A}[\![Q[e/v]]\!](s,h)(s',h') = \mathcal{A}[\![Q]\!](s[v \mapsto \mathcal{E}[\![e]\!](s,h)],h)(s',h')\ .$$

Proof. The proof proceeds by structural induction on Q. We first prove the claim for an arbitrary logical expression q by structural induction on q.

We first consider the important case where $q \equiv v$. Now first assume that $[e] \prec v$. Then $[v]$ must be some reference type C because the subtype relation does not allow subtypes of primitive types. We then compute as follows.

$$\begin{aligned}
&\mathcal{L}[\![v[e/v]]\!](s,h)(s',h') \\
&= \mathcal{L}[\![([v])e]\!](s,h)(s',h') && \{\text{ def. } [e/v]\ \} \\
&= \mathcal{L}[\![e]\!](s,h)(s',h') && \{\ [e] \preceq [v] \text{ and Lemma 4.1 }\} \\
&= \mathcal{E}[\![e]\!](s,h) && \{\text{ Lemma 3.2 }\} \\
&= \mathcal{L}[\![v]\!](s[v \mapsto \mathcal{E}[\![e]\!](s,h)],h)(s',h') && \{\text{ def. } \mathcal{L}[\![_]\!]\ \}
\end{aligned}$$

[1] Dijkstra [Dij76] used the term weakest *liberal* precondition for what we call the weakest precondition here. He reserved the latter term for the formula that additionally guarantees that execution of the statement terminates. For brevity, we will continue to use the term weakest precondition, but we ask the reader to bear in mind that what we mean is actually the weakest liberal precondition.

If $[e] = [v]$ we immediately get $\mathcal{L}[\![v[e/v]]\!](s,h)(s',h') = \mathcal{L}[\![e]\!](s,h)(s',h')$ from the definition of $[e/v]$. The rest of the above computation remains the same.

The only other interesting case concerns a logical expression of the form $q.x$. Let $\mathcal{L}[\![q[e/v]]\!](s,h)(s',h') \neq \mathit{null}$.

$$
\begin{array}{ll}
\mathcal{L}[\![q.x[e/v]]\!](s,h)(s',h') & \\
= \mathcal{L}[\![(q[e/v]).x]\!](s,h)(s',h') & \{ \text{ def. } [e/v] \} \\
= h(\mathcal{L}[\![q[e/v]]\!](s,h)(s',h'))(\mathsf{origin}([q[e/v]],x))(x) & \{ \text{ def. } \mathcal{L}[\![\_]\!] \} \\
= h(\mathcal{L}[\![q[e/v]]\!](s,h)(s',h'))(\mathsf{origin}([q],x))(x) & \{ \text{ Lemma 4.2 } \} \\
= h(\mathcal{L}[\![q]\!](s[v \mapsto \mathcal{E}[\![e]\!](s,h)],h)(s,h))(\mathsf{origin}([q],x))(x) & \{ \text{ ind. hyp. } \} \\
= \mathcal{L}[\![q.x]\!](s[v \mapsto \mathcal{E}[\![e]\!](s,h)],h)(s',h') & \{ \text{ def. } \mathcal{L}[\![\_]\!] \}
\end{array}
$$

It is not difficult to show that the claim also holds if $\mathcal{L}[\![q[e/v]]\!](s,h)(s',h') = \mathit{null}$. Both expressions evaluate to $\perp$ in that case. □

The theorem below reveals that $\mathsf{defined}(e) \to Q[e/u]$ is a valid precondition of Q with respect to the assignment $u := e$.

Theorem 4.4. *For every assertion Q, and every expression e and local variable u such that $[e] \preceq [u]$, we have*

$$\models \{\mathsf{defined}(e) \to Q[e/u]\}\ u := e\ \{Q\}\ .$$

Proof. Let $(s,h)(s',h') \models \mathsf{defined}(e) \to Q[e/u]$. Let $\langle u := e, (s,h)\rangle \to (s'',h'')$. This computation can only be derived using rule LA of the operation semantics. Hence the assumption of that rule must also hold. This assumption states that $\mathcal{E}[\![e]\!](s,h) \neq \perp$, which implies that $(s,h)(s',h') \models \mathsf{defined}(e)$. Hence $(s,h)(s',h') \models Q[e/u]$. Moreover, we have $s'' = s[u \mapsto \mathcal{E}[\![e]\!](s,h)]$ and $h'' = h$. Then Lemma 4.3 implies that $(s'',h)(s',h') \models Q$. □

The final bit of support for our claim regarding the weakest precondition of an assignment $u := e$ with respect to an assertion is provided by the theorem below. In fact, the theorem states that it always suffices to check the validity of the assertion $Q \to (\mathsf{defined}(e) \to Q'[e/u])$ if we want to check the validity of a Hoare triple $\{Q\}\ u := e\ \{Q'\}$. Thus it ensures that we can replace a proof obligation $\{Q\}\ u := e\ \{Q'\}$ by the verification condition $Q \to (\mathsf{defined}(e) \to Q'[e/u])$.

Theorem 4.5. *Let Q, Q' be arbitrary assertions. Let e be an expression, and let u be a local variable such that $[e] \preceq [u]$. Then*

$$\models \{Q\}\ u := e\ \{Q'\} \quad \Longleftrightarrow \quad \models Q \to (\mathsf{defined}(e) \to Q'[e/u])\ .$$

Proof. We first prove that $\models \{Q\}\ u := e\ \{Q'\}$ implies that our verification condition $Q \to (\mathsf{defined}(e) \to Q'[e/u])$ holds. Let $(s,h)(s',h') \models Q$. If $\mathcal{E}[\![e]\!](s,h) = \perp$, then $(s,h)(s,h') \not\models \mathsf{defined}(e)$. Assume therefore that $\mathcal{E}[\![e]\!](s,h) = v \neq \perp$. Then

$$\langle u := e, (s,h)\rangle \to (s[u \mapsto v],h)$$

according to rule LA of the operation semantics. From $\models \{Q\}\ u := e\ \{Q'\}$ and $(s,h)(s',h') \models Q$ we may then conclude that $(s[u \mapsto v],h)(s',h') \models Q'$. Lemma 4.3 then implies that $(s,h)(s',h') \models Q'[e/u]$.

To prove the implication in the opposite direction, assume $(s,h)(s',h') \models Q$, and the existence of a computation $\langle u := e,(s,h)\rangle \to (s'',h'')$. By the validity of $Q \to (\mathsf{defined}(e) \to Q'[e/u])$ we get $(s,h)(s',h') \models \mathsf{defined}(e) \to Q'[e/u]$. The assumption of rule LA, which much have been used to derive the above computation, states that $\mathcal{E}[\![e]\!](s,h) \neq \bot$. Then $(s,h)(s',h') \models \mathsf{defined}(e)$. Hence $(s,h)(s',h') \models Q'[e/u]$. Moreover, in the context of LA we have $(s'',h'') = (s[u \mapsto v],h)$. Then Lemma 4.3 implies $(s'',h'')(s',h') \models Q'$. □

We can use similar techniques to reason about local variable declarations. We already argued above that a variable declaration statement $t\ u$ can be viewed as an assignment $u := \mathsf{def}(t)$, where $\mathsf{def}(t)$ denotes an expression whose value is the default value of type t in all states. The expression $\mathsf{def}(t)$ is defined for every program type t by the following equations.

$$\begin{aligned} \mathsf{def}(\mathsf{boolean}) &= \mathsf{false} \\ \mathsf{def}(\mathsf{int}) &= 0 \\ \mathsf{def}(C) &= \mathsf{null} \end{aligned}$$

In fact, reasoning about local variable allocation is simpler than reasoning about arbitrary local assignments because the value of $\mathsf{def}(t)$ is never equal to $\bot$. Hence $Q[\mathsf{def}(t)/u]$ is the weakest precondition of a statement $t\ u$ with respect to an assertion Q. We finish this section with two theorems that support this claim.

Theorem 4.6. *For every assertion Q, and every local variable declaration $t\ u$ we have*

$$\models \{Q[\mathsf{def}(t)/u]\}\ t\ u\ \{Q\}\ .$$

Proof. A simple variant of the proof of Theorem 4.4 using rule (*VA*) of the operational semantics. □

Theorem 4.7. *Let Q, Q' be arbitrary assertions. Then*

$$\models \{Q\}\ t\ u\ \{Q'\} \quad \Longleftrightarrow \quad \models Q \to (Q'[\mathsf{def}(t)/u])\ .$$

for every valid program type t and every local variable u.

Proof. Along the lines of the proof of Theorem 4.5. □

4.2 Field Assignments and Aliasing

A statement of the form $e.x := e'$ assigns the value of e' to field x of the object that is referenced by e. Reasoning about field assignments is more complex than reasoning about local assignments because the expression e that denotes

the object may have aliases. That is, there may be other expressions e' that reference the same object.

It is not possible, in general, to determine statically if two access expressions $e.x$ and $e'.x$ denote the same heap location. Recall from Sect. 2.2.3 that $\mathsf{origin}([e], x)$ denotes the class in which the field x of an access expression $e.x$ is declared. The access expressions $e.x$ and $e'.x$ denote the same heap location if e and e' reference the same object, and if $\mathsf{origin}([e], x)$ is equal to $\mathsf{origin}([e'], x)$. The last condition, which is necessary due to field shadowing, can be resolved statically since the static types of the expression can be derived statically. However, it is in general not possible to resolve the first condition statically because one cannot determine beforehand if two expressions will denote the same object at run-time.

In special circumstances one can determine that two expressions e and e' cannot be aliases by inspecting their static types. The expressions cannot be aliases of the same object if the types of the two expressions are unrelated. Lemma 2.3 states that the union of the domains of two unrelated types C and D is the singleton set $\{null\}$. Moreover, we know from Lemma 3.1 that evaluating an expression with type C always yields an element from the domain of C if the value of the expression is defined. Hence we can prove the following lemma.

Lemma 4.8. *Let q, q' be two expressions of a reference type. Let (s, h) be an arbitrary state. Let (s', h') be a compatible freeze state. If $[q] \not\sim [q']$ then*

$$\mathcal{L}[\![q]\!](s,h)(s',h') = \mathcal{L}[\![q']\!](s,h)(s',h') \Longrightarrow \mathcal{L}[\![q]\!](s,h)(s',h') \in \{null, \bot\} \ .$$

Proof. A simple case analysis using Lemma 2.3 and Lemma 3.1. □

The above observations can be used to define a syntactical substitution operation $[e'/e.x]$ that substitutes all aliases of $e.x$ by e'. Obviously, the most interesting case of this substitution operation is $q.x[e'/e.x]$ because $q.x$ may be an alias of the expression $e.x$. The substitution should replace $q.x$ by e' if it is an alias of $e.x$. We define $q.x[e'/e.x]$ using a conditional expression if we cannot statically guarantee that $q.x$ is not an alias of $e.x$. Let $q' \equiv q[e'/e.x]$. Then we define $q.x[e'/e.x]$ as follows.

$$q.x[e'/e.x] \quad \equiv \quad \begin{cases} q'.x & \text{if } [q] \not\sim [e] \text{ or} \\ & \mathsf{origin}([q], x) \neq \mathsf{origin}([e], x) \\ q' = e \ ? \ ([q.x])e' : q'.x & \text{otherwise} \end{cases} \tag{4.1}$$

The cast in $([q.x])e'$ can be omitted if $q.x$ and e' have the same static type. We have $q.y[e'/e.x] \equiv (q[e'/e.x]).y$ for every field y that is not equal to x. Moreover, $u[e'/e] \equiv u$ for all local variables u. All other cases of $q[e'/e.x]$ are defined in the same way as $q[e/u]$ in Fig. 4.1.

Example 4.2. *Figure 4.2 shows a class that models a clock. Clocks have a sync-method that synchronizes the time of the clock with that of a master clock in*

```
class Clock {
  int time ;

  requires ¬(this = m);
  ensures this.time = m.time − 1;
  void sync(Clock m) {
    assert ¬(this = m);
    this.time := m.time − 1 ;
    assert this.time = m.time − 1;
  }
}
```

Figure 4.2: An example proof outline that involves reasoning about aliases.

such a way that all clocks lag one tick behind the master clock. The master clock is passed as a parameter to the method. The precondition of the sync method requires that the clock that is synchronized is not the master clock itself. We can show that this requirement is necessary by considering the proof obligations of the method. The proof obligation that corresponds to the assignment in the body of the method is the Hoare triple

$$\{\neg(\mathsf{this} = m)\}\ \mathsf{this}.time := m.time - 1\ \{\mathsf{this}.time = m.time - 1\}\ .$$

We will show that the precondition $\neg(\mathsf{this} = m)$ *is necessary to prove*

$$\mathsf{this}.time = m.time - 1[m.time{-}1/\mathsf{this}.time]\ . \tag{4.2}$$

In order to prove our claim we first compute

$$(\mathsf{this}.time = m.time - 1)[m.time{-}1/\mathsf{this}.time],$$

which is by the definition of the substitution operation equal to

$$((\mathsf{this}.time)[m.time{-}1/\mathsf{this}.time]) = ((m.time - 1)[m.time{-}1/\mathsf{this}.time]).$$

The left-hand side of this equality is computed as follows.

$$\begin{aligned}
&(\mathsf{this}.time)[m.time{-}1/\mathsf{this}.time]\\
&= (\mathsf{this}[m.time{-}1/\mathsf{this}.time]) = \mathsf{this}\ ?\ m.time{-}1\\
&\qquad : (\mathsf{this}[m.time{-}1/\mathsf{this}.time]).time\\
&= (\mathsf{this} = \mathsf{this}\ ?\ m.time - 1 : \mathsf{this}.time)
\end{aligned}$$

The right-hand side is computed similarly.

$$\begin{aligned}
&(m.time - 1)[m.time{-}1/\mathsf{this}.time]\\
&= ((m.time[m.time{-}1/\mathsf{this}.time]) - (1[m.time{-}1/\mathsf{this}.time])\\
&= ((m[m.time{-}1/\mathsf{this}.time]) = \mathsf{this}\ ?\ m.time{-}1\\
&\qquad : (m[m.time{-}1/\mathsf{this}.time]).time){-}1\\
&= (m = \mathsf{this}\ ?\ m.time - 1 : m.time) - 1
\end{aligned}$$

By combining the two resulting formulas we obtain the formula

$$(\mathsf{this} = \mathsf{this}\ ?\ m.time - 1 : \mathsf{this}.time) = (m = \mathsf{this}\ ?\ m.time - 1 : m.time) - 1,$$

which is equivalent to

$$(m.time - 1) = (m = \mathsf{this}\ ?\ m.time - 1 : m.time) - 1.$$

This formula clearly only holds if the value of m differs from the value of this*. Hence our precondition was necessary. This example shows how the substitution operation for field assignments reveals possible aliases to the reasoner.*

It is not difficult to see that $[e'/e.x]$ is a type-preserving substitution operation. In (4.1), we have that the type of $([q.x])e'$ is equal to $[q.x]$. The type of the access expression $q'.x$ in (4.1) depends on the type of q'. By the induction hypothesis, we have $[q'] = [q]$. Hence $[q'.x] = [q.x]$. Therefore the following lemma holds.

Lemma 4.9. *For every logical expression q, every access expression $e.x$, and every expression e' such that $[e'] \preceq [e.x]$, we have $[q[e'/e.x]] = [q]$.*

Proof. By structural induction on q. □

We can also prove that $Q[e'/e.x]$ predicts if Q holds after assigning the value of e' to $e.x$.

Lemma 4.10. *Let $e.x := e'$ a well-typed assignment. Let $\mathsf{origin}([e], x) = D$. Let (s, h) be a state such that $\mathcal{E}[\![e]\!](s, h) = o \notin \{\bot, null\}$ and $\mathcal{E}[\![e']\!](s, h) = v \neq \bot$. Let (s', h') be a compatible freeze state. Then*

$$\mathcal{A}[\![Q[e'/e.x]]\!](s, h)(s', h') = \mathcal{A}[\![Q]\!](s, h[o.x_D \mapsto v])(s, h')\ ,$$

where $h[o.x_D \mapsto v]$ denotes the heap that results from h by assigning v to the field x of object o that is declared in class D.

Proof. We prove the lemma by structural induction on Q. For the base case we must prove, by structural induction on q, that

$$\mathcal{L}[\![q[e'/e.x]]\!](s, h)(s', h') = \mathcal{L}[\![q]\!](s, h[o.x_D \mapsto v])(s', h')\ , \tag{4.3}$$

holds for every logical expression q.

Let $h^* = h[o.x_D \mapsto v]$. The only interesting case of (4.3) concerns a logical expression of the form $q.x$. We must prove that

$$\mathcal{L}[\![q.x[e'/e.x]]\!](s, h)(s', h') = \mathcal{L}[\![q.x]\!](s, h^*)(s', h')\ . \tag{4.4}$$

follows from the induction hypothesis

$$\mathcal{L}[\![q[e'/e.x]]\!](s, h)(s', h') = \mathcal{L}[\![q]\!](s, h^*)(s', h')\ .$$

Note that (4.4) holds trivially if $\mathcal{L}[\![q[e'/e.x]]\!](s,h)(s',h') \in \{\mathsf{null}, \bot\}$. We will therefore assume that $\mathcal{L}[\![q[e'/e.x]]\!](s,h)(s',h')$ is not contained in $\{\mathsf{null}, \bot\}$ in the rest of this proof.

If $\mathsf{origin}([q], x) = E \neq D$ we get the following computation.

$$
\begin{array}{ll}
& \mathcal{L}[\![q.x[e'/e.x]]\!](s,h)(s',h') \\
= & \mathcal{L}[\![(q[e'/e.x]).x]\!](s,h)(s',h') & \{ \text{ def. } [e'/e.x] \} \\
= & h(\mathcal{L}[\![q[e'/e.x]]\!](s,h)(s',h'))(\mathsf{origin}([q[e'/e.x]], x))(x) & \{ \text{ def. } \mathcal{L}[\![\_]\!] \} \\
= & h(\mathcal{L}[\![q[e'/e.x]]\!](s,h)(s',h'))(E)(x) & \{ \text{ Lemma 4.9 } \} \\
= & h(\mathcal{L}[\![q]\!](s,h^*)(s',h'))(E)(x) & \{ \text{ ind. hyp. } \} \\
= & h^*(\mathcal{L}[\![q]\!](s,h^*)(s',h'))(E)(x) & \{ \ E \neq D \ \} \\
= & \mathcal{L}[\![q.x]\!](s,h^*)(s,h') & \{ \text{ def. } \mathcal{L}[\![\_]\!] \}
\end{array}
$$

In the sequel, we will assume that $\mathsf{origin}([q], x) = D$. Next, we prove that (4.4) holds if $[q] \not\sim [e]$. By the contraposition of Lemma 4.8 and our previous assumptions we then get

$$\mathcal{L}[\![q]\!](s,h)(s',h') \neq \mathcal{L}[\![e]\!](s,h)(s',h') = \mathcal{E}[\![e]\!](s,h) = o \ . \tag{4.5}$$

The second step follows from Lemma 3.2. The following proof shows that (4.4) holds if $[q] \not\sim [e]$.

$$
\begin{array}{ll}
& \mathcal{L}[\![q.x[e'/e.x]]\!](s,h)(s',h') \\
= & \mathcal{L}[\![(q[e'/e.x]).x]\!](s,h)(s',h') & \{ \text{ def. } [e'/e.x] \} \\
= & h(\mathcal{L}[\![q[e'/e.x]]\!](s,h)(s',h'))(\mathsf{origin}([q[e'/e.x]], x))(x) & \{ \text{ def. } \mathcal{L}[\![\_]\!] \} \\
= & h(\mathcal{L}[\![q[e'/e.x]]\!](s,h)(s',h'))(D)(x) & \{ \text{ Lemma 4.9 } \} \\
= & h(\mathcal{L}[\![q]\!](s,h^*)(s',h'))(D)(x) & \{ \text{ ind. hyp. } \} \\
= & h^*(\mathcal{L}[\![q]\!](s,h^*)(s',h'))(D)(x) & \{ \text{ Eq. 4.5 } \} \\
= & \mathcal{L}[\![q.x]\!](s,h^*)(s',h') & \{ \text{ def. } \mathcal{L}[\![\_]\!] \}
\end{array}
$$

For the final case we assume that $\mathsf{origin}([q], x) = D$ and $[q] \sim [e]$. By applying the definition of $[e'/e.x]$ we then get

$$
\begin{array}{l}
\mathcal{L}[\![q.x[e'/e.x]]\!](s,h)(s',h') \\
\qquad = \mathcal{L}[\![(q[e'/e.x]) = e \ ? \ ([q.x])e' : (q[e'/e.x]).x]\!](s,h)(s',h') \ .
\end{array}
$$

If $\mathcal{L}[\![q[e'/e.x]) = e]\!](s,h)(s',h')$ does not evaluate to true, we know that Eq. (4.5) holds again, and we can finish the proof along the lines of the previous case. Therefore assume that $\mathcal{L}[\![q[e'/e.x]) = e]\!](s,h)(s',h') = \mathsf{true}$. We then have

$$
\begin{array}{l}
\mathcal{L}[\![(q[e'/e.x]) = e \ ? \ ([q.x])e' : (q[e'/e.x]).x]\!](s,h)(s',h') \\
\qquad\qquad = \mathcal{L}[\![([q.x])e']\!](s,h)(s',h')
\end{array}
$$

by the definition of $\mathcal{L}[\![\_]\!]$. Moreover, $\mathcal{L}[\![q[e'/e.x])]\!](s,h)(s',h')$ must be equal to $\mathcal{L}[\![e]\!](s,h)(s',h')$, which by Lemma 3.2 implies

$$\mathcal{L}[\![q[e'/e.x])]\!](s,h)(s',h') = \mathcal{L}[\![e]\!](s,h)(s',h') \ . \tag{4.6}$$

The final case is then computed as follows.

$$
\begin{array}{ll}
\mathcal{L}[\![([q.x])e']\!](s,h)(s',h') & \\
= \mathcal{L}[\![e']\!](s,h)(s',h') & \{ \text{ def. } \mathcal{L}[\![\_]\!],\ [e'] \preceq [l.x],\ \text{Lemma 3.1 } \} \\
= h^*(o)(D)(x) & \{ \text{ def. } h^* \ \} \\
= h^*(\mathcal{L}[\![q[e'/e.x]]\!](s,h)(s',h'))(D)(x) & \{ \text{ Eq. (4.6) } \} \\
= h^*(\mathcal{L}[\![q]\!](s,h^*)(s',h'))(D)(x) & \{ \text{ ind. hyp. } \} \\
= \mathcal{L}[\![q.x]\!](s,h^*)(s',h') & \{ \text{ def. } \mathcal{L}[\![\_]\!] \ \}
\end{array}
$$

□

The lemma we just proved shows that $Q[e'/e.x]$ predicts if Q holds after an assignment $e.x := e'$. Therefore it can be used to describe the weakest precondition of an assignment of that form with respect to an assertion Q. The complete weakest precondition of an assignment $e.x := e'$ with respect to an assertion Q is the assertion $Q[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \vee \neg\mathsf{defined}(e')$. We will prove this claim along the same lines as we did in the previous section for local assignments. That is, we first prove that the given assertion is a valid precondition.

Theorem 4.11. *For every assertion Q, and every well-typed assignment of the form $e.x := e'$, we have*

$$\models \{Q[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \vee \neg\mathsf{defined}(e')\}\ e.x := e'\ \{Q\}\ .$$

Proof. Let (s,h) be a state such that $\langle e.x := e', (s,h)\rangle \to (s'',h'')$. Let (s',h') be a compatible freeze state such that

$$(s,h)(s',h') \models Q[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \vee \neg\mathsf{defined}(e')\ .$$

The computation guarantees that $\mathcal{E}[\![e]\!](s,h) = o \notin \{\bot, \mathsf{null}\}$ and $\mathcal{E}[\![e']\!](s,h) = v \neq \bot$ according to rule *FA* of the operational semantics. Moreover, we have $s'' = s$ and $h'' = h[o.x_D \mapsto v]$, where $D = \mathsf{origin}([e], x)$. Then we also have $(s,h)(s',h') \not\models \neg\mathsf{defined}(e)$ and $(s,h)(s',h') \not\models \neg\mathsf{defined}(e')$. From $o \neq \mathsf{null}$ follows $(s,h)(s',h') \not\models e = \mathsf{null}$. Hence $(s,h)(s',h') \models Q[e'/e.x]$. Then Lemma 4.10 implies that $(s'',h'')(s',h') \models Q$. □

Secondly, we prove that the give precondition is indeed the weakest precondition.

Theorem 4.12. *Let Q, Q' be arbitrary assertions. Let $e.x := e'$ be a well-typed assignment. Then*

$$\begin{array}{c} \models \{Q\}\ e.x := e'\ \{Q'\} \\ \Longleftrightarrow \\ \models Q \to (Q'[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \vee \neg\mathsf{defined}(e'))\ . \end{array}$$

Proof. We first prove that $\models \{Q\}\ e.x := e'\ \{Q'\}$ implies

$$\models Q \rightarrow (Q'[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \vee \neg\mathsf{defined}(e')\ .$$

Let $(s,h)(s',h') \models Q$. If $\mathcal{E}[\![e]\!](s,h) = \bot$ or $\mathcal{E}[\![e']\!](s,h) = \bot$ then we have $(s,h)(s,h') \models \neg\mathsf{defined}(e)$ or $(s,h)(s',h') \models \neg\mathsf{defined}(e')$. If $\mathcal{E}[\![e]\!](s,h) = \mathit{null}$ then $(s,h)(s',h') \models e = \mathsf{null}$. Assume therefore that $\mathcal{E}[\![e]\!](s,h) = o \notin \{\bot, \mathit{null}\}$ and that $\mathcal{E}[\![e']\!](s,h) = v \neq \bot$. Then

$$\langle e.x := e', (s,h)\rangle \rightarrow (s, h[o.x_D \mapsto v])$$

according to rule *FA* of the operation semantics, where $D = \mathsf{origin}([e], x)$. From $\models \{Q\}\ e.x := e'\ \{Q'\}$ and $(s,h)(s',h') \models Q$ we may then conclude that $(s, h[o.x_D \mapsto v])(s',h') \models Q'$. Then $(s,h)(s',h') \models Q[e'/e.x]$ follows from Lemma 4.10.

To prove the implication in the opposite direction, assume $(s,h)(s',h') \models Q$, and the existence of a computation $\langle e.x := e', (s,h)\rangle \rightarrow (s'',h'')$. By the validity of the right-hand side we then infer

$$(s,h)(s',h') \models Q'[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \vee \neg\mathsf{defined}(e')\ .$$

From rule *FA* we get $\mathcal{E}[\![e]\!](s,h) = o \notin \{\bot, \mathit{null}\}$ and $\mathcal{E}[\![e']\!](s,h) \neq \bot$. Moreover, we have $(s'',h'') = (s, h[o.x_D \mapsto v])$, where $D = \mathsf{origin}([e], x)$. Then it must be the case that $(s,h)(s',h') \models Q'[e'/e.x]$. Lemma 4.10 then states that $(s'',h'')(s',h') \models Q'$. □

4.3 Strongest Postconditions

After having described the weakest preconditions of assignments in the previous section, we will use this section to present their strongest postconditions. Weakest preconditions are usually more concise than strongest postconditions because assertions of the latter type must contain all information that may be relevant in the final state, whereas weakest preconditions are, as the name weakest precondition already suggests, minimal formulas that ensure that the desired postconditions hold in the final state.

Our main reason for studying the strongest postconditions of assignments is that the standard pattern for proving completeness of procedures with parameters [Gor75] assumes that the strongest postconditions of arbitrary statements can be expressed in the assertion language. This assumption can only hold if we are also able to express the strongest postconditions of assignments in our assertion language. Hence we will investigate the strongest postconditions of assignments below.

We will denote the strongest postcondition of a statement S with respect to an assertion Q by $\mathsf{sp}(Q,S)$. Formally, $\mathsf{sp}(Q,S)$ is an assertion such that $\models \{Q\}\ S\ \{\mathsf{sp}(Q,S)\}$, and $\models \{Q\}\ S\ \{Q'\}$ implies $\models \mathsf{sp}(Q,S) \rightarrow Q'$. There may be several syntactically distinct (but equivalent) formulas that satisfy these two

criteria, so it would actually be more correct to speak of *a* strongest postcondition instead of *the* strongest postcondition. Naturally, each strongest postcondition is equivalent to every other strongest postcondition.

The following equations describe strongest postconditions for local assignments.

$$\begin{array}{rcl} \mathsf{sp}(Q, u := e) & = & (\exists z : [u] \bullet (Q \wedge \mathsf{defined}(e))[z/u] \wedge u = (e[z/u])) \\ \mathsf{sp}(Q, t\ u) & = & (\exists z : t \bullet Q[z/u]) \wedge u = \mathsf{def}(t) \end{array}$$

Strictly speaking, we have not (yet) defined the substitution operation $[z/u]$ because the logical variable z is not a program expression. There is a well-known problem if we allow logical variables in the substitution operation because replacing free occurrences of a variable by a logical variable may result in a formula in which the latter is bound. This happens if the program variable occurs in the scope of a quantifier for the logical variable. We will therefore assume that any logical variable z that is introduced to express a strongest postcondition is fresh, i.e., that it does not occur in Q. It is not difficult to prove that this assumption ensures that Lemma 4.3 also holds for the substitution $[z/u]$.

The definition of the strongest postcondition of local assignments is fairly standard. But the strongest postcondition of field assignments is a different story: we are not aware of previous attempts to formulate the strongest postcondition of arbitrary field assignments of the form $e.x := e'$. We propose the following strongest postcondition for an assignment $e.x := e'$ with respect to an arbitrary assertion Q.

$$\begin{array}{l} (\exists o : [e] \bullet \exists z : [e.x] \bullet (Q \wedge \mathsf{defined}(e) \wedge \neg(e = \mathsf{null}) \wedge \mathsf{defined}(e'))[z/o.x] \\ \qquad\qquad \wedge\ o = (e[z/o.x]) \wedge o.x = (e'[z/o.x])) \end{array} \quad (4.7)$$

The logical variables o and z must be fresh. The substitution operation $[z/o.x]$ is the obvious variant of $[e'/e.x]$ with the program expressions e and e' replaced by logical variables.

The strongest postcondition describes the resulting state if the assignment $e.x := e'$ is successfully executed in a state that satisfies the precondition Q. It says that there exist an object o and a value z such that the precondition Q holds if we set the value of $o.x$ (back) to z. This clearly holds if we choose the old values of e and $e.x$ for o and z, respectively. The state change is modelled by the substitution $[z/o.x]$. The old state also satisfies

$$\mathsf{defined}(e) \wedge \neg(e = \mathsf{null}) \wedge \mathsf{defined}(e')$$

because the assignment would have gone wrong otherwise. The second line of the strongest postcondition says that the object o is the old value of e, and that the new value of $o.x$ is the old value of e'.

We can prove the following substitution lemma for the operation $[z/o.x]$.

Lemma 4.13. *Let o and z be logical variables such that $[o.x] = [z]$. Let Q be an arbitrary assertion in which o and z do not occur. Let (s, h) be a state such that $s(o) = \alpha \neq null$, and let (s', h') be a compatible freeze state. Then*

$$\mathcal{A}[\![Q[z/o.x]]\!](s,h)(s',h') = \mathcal{A}[\![Q]\!](s, h[\alpha.x_D \mapsto \beta])(s',h') \ ,$$

where $D = \mathsf{origin}([o], x)$, and $\beta = s(z)$.

Proof. By structural induction on Q, along the lines of the proof of Lemma 4.10. □

The following lemma plays an important role in the justification of our strongest postcondition. It says that we may substitute a logical variable for an access expression if we assign the value of the access expression to the logical variable.

Lemma 4.14. *Let o and z be logical variables such that $[o.x] = [z]$. Let Q be an arbitrary assertion in which o and z do not occur. Let (s, h) be a state such that $s(o) = \alpha \neq null$, and let (s', h') be a compatible freeze state. Then*

$$\mathcal{A}[\![Q]\!](s,h)(s',h') = \mathcal{A}[\![Q[z/o.x]]\!](s[z \mapsto h(\alpha)(D)(x)], h)(s',h')$$

where $D = \mathsf{origin}([o], x)$.

Proof. By structural induction on Q. □

There is another useful property of the substitution operation $[z/o.x]$: it makes an assertion insensitive to other substitutions of the same field.

Lemma 4.15. *Let Q be an arbitrary assertion. Let $e.x := e'$ be a well-typed assignment. Let o and z be arbitrary logical variables such that $[o] = [e]$ and $[z] = [e.x]$. Then*

$$(s,h)(s',h') \models Q[z/o.x][e'/e.x] \iff (s,h)(s',h') \models Q[z/o.x]$$

for every current state (s, h), and every compatible freeze state (s', h') such that $s(o) = \mathcal{E}[\![e]\!](s, h)$.

Proof. By structural induction on Q. □

The following theorem justifies our strongest postcondition of field assignments.

Theorem 4.16. *For every well-typed assignment $e.x := e'$, and every precondition Q, we have that $\mathsf{sp}(Q, e.x := e')$ as defined in Eq. 4.7 denotes a strongest postcondition of $e.x := e'$ with respect to Q.*

Proof. We first prove $\models \{Q\}\ e.x := e'\{\mathsf{sp}(Q, e.x := e')\}$. Let $(s,h)(s',h') \models Q$ and $\langle e.x := e', (s,h)\rangle \to (s,h'')$. Rule *FA* of the operational semantics ensures that $\mathcal{E}[\![e]\!](s,h) = \alpha \notin \{\bot, \mathsf{null}\}$ and $\mathcal{E}[\![e']\!](s,h) = v \neq \bot$. Moreover, we know that $h'' = h[\alpha.x_C \mapsto v]$, where $C = \mathsf{origin}([e], x)$. We must prove that $(s,h'')(s',h') \models \mathsf{sp}(Q, e.x := e')$.

Let $\beta = \mathcal{E}[\![e.x]\!](s,h)$. Note that $\beta \neq \bot$ because $\mathcal{E}[\![e]\!](s,h) \notin \{\bot, \mathsf{null}\}$. Let $s^* = s[o, z \mapsto \alpha, \beta]$. We first prove that $(s^*,h'')(s',h') \models Q'[z/o.x]$, where Q' abbreviates the formula $Q \wedge \mathsf{defined}(e) \wedge \neg(e = \mathsf{null}) \wedge \mathsf{defined}(e')$.

From $(s,h)(s',h') \models Q$, $\alpha \notin \{\bot, \mathit{null}\}$ and $v \neq \bot$ follows $(s,h)(s',h') \models Q'$. We also have $(s^*,h)(s',h') \models Q'$ because o and z do not occur in Q'. Hence $(s^*,h)(s',h') \models Q'[z/o.x]$ by Lemma 4.14. The original substitution lemma (Lemma 4.10) then yields $(s^*,h'')(s',h') \models Q'[z/o.x][e'/e.x]$. The latter formula is equivalent to $(s^*,h'')(s',h') \models Q'[z/o.x]$ according to Lemma 4.15.

Secondly, we will prove that $(s^*,h'')(s',h') \models o = (e[z/o.x])$. Note that

$$\mathcal{L}[\![o]\!](s^*,h'')(s',h') = s^*(o) = \alpha = \mathcal{E}[\![e]\!](s,h) = \mathcal{E}[\![e]\!](s^*,h)\ .$$

The last step is valid because o and z do not occur in e. Moreover, we have

$$\begin{array}{ll}
\mathcal{E}[\![e]\!](s^*,h) & \\
= \mathcal{L}[\![e]\!](s^*,h)(s',h') & \{\text{ Lemma 3.2 }\} \\
= \mathcal{L}[\![e[z/o.x]]\!](s^*,h)(s',h') & \{\text{ Lemma 4.14 }\} \\
= \mathcal{L}[\![e[z/o.x][e'/e.x]]\!](s^*,h'')(s',h') & \{\text{ Lemma 4.10 }\} \\
= \mathcal{L}[\![e[z/o.x]]\!](s^*,h'')(s',h')\ . & \{\text{ Lemma 4.15 }\}
\end{array}$$

Note also that $\alpha \neq \bot$. Hence $(s^*,h'')(s',h') \models o = (e[z/o.x])$.

Finally, we must prove that $(s^*,h'')(s',h') \models o.x = (e'[z/o.x])$. We have

$$\mathcal{L}[\![o.x]\!](s^*,h'')(s',h') = h''(s^*(o))(C)(x) = h''(\alpha)(C)(x) = v = \mathcal{E}[\![e']\!](s,h)$$

by the construction of s^* and h''. Clearly, $\mathcal{E}[\![e']\!](s,h) = \mathcal{E}[\![e']\!](s^*,h)$ because o and z do not occur in e'. Along the same steps as above we can show that $\mathcal{E}[\![e']\!](s^*,h) = \mathcal{L}[\![e'[z/o.x]]\!](s^*,h'')(s',h')$. The previous subproofs collectively imply that $(s^*,h'')(s',h') \models o.x = (e'[z/o.x])$, which completes our proof of $(s,h)(s',h') \models \mathsf{sp}(Q, e.x := e')$.

The second half of the entire proof must show that $\models \{Q\}\ e.x := e'\ \{Q'\}$ implies $\models \mathsf{sp}(Q, e.x := e') \to Q'$. So let $(s,h)(s',h') \models \mathsf{sp}(Q, e.x := e')$. Then there exist values α and β such that

$$\begin{aligned}(s^*,h)(s',h') \models (Q \wedge \mathsf{defined}(e) \wedge \neg(e = \mathsf{null}) \wedge \mathsf{defined}(e'))[z/o.x] \\ \wedge\ o = (e[z/o.x]) \wedge o.x = (e'[z/o.x])\ ,\end{aligned} \quad (4.8)$$

where $s^* = s[o, z \mapsto \alpha, \beta]$. We will prove that

$$\langle e.x := e', (s^*, h[\alpha.x_C \mapsto \beta])\rangle \to (s^*, h)\ , \quad (4.9)$$

where $C = \mathsf{origin}([o], x)$, can be derived using rule *FA* of the operational semantics. Let $v_1 = \mathcal{E}[\![e]\!](s^*, h[\alpha.x_C \mapsto \beta])$, and $v_2 = \mathcal{E}[\![e']\!](s^*, h[\alpha.x_C \mapsto \beta])$. Observe

that $\mathsf{origin}([e], x) = \mathsf{origin}([o], x) = C$ because $[o] = [e]$. Then we can derive the above computation if the following conditions hold.

- $h[\alpha.x_C \mapsto \beta][v_1.x_C \mapsto v_2] = h$
- $v_1 \notin \{\bot, null\}$
- $v_2 \neq \bot$.

Note that the first condition holds if $\alpha = v_1$ and $v_2 = h(v_1)(C)(x)$. We have

$$\begin{array}{ll} \alpha = s^*(o) & \{ \text{ def. } s^* \ \} \\ \quad = \mathcal{L}[\![e[z/o.x]]\!](s^*, h)(s', h') & \{ \text{ (Eq. 4.8) } \} \\ \quad = \mathcal{E}[\![e]\!](s^*, h[\alpha.x_C \mapsto \beta]) & \{ \text{ Lemma 4.13 } \} \\ \quad = v_1 \ . & \end{array}$$

Moreover, we have

$$\begin{array}{ll} v_2 = \mathcal{E}[\![e']\!](s^*, h[\alpha.x_C \mapsto \beta]) & \\ \quad = \mathcal{L}[\![e']\!](s^*, h[\alpha.x_C \mapsto \beta])(s', h') & \{ \text{ Lemma 3.2 } \} \\ \quad = \mathcal{L}[\![e'[z/o.x]]\!](s^*, h)(s', h') & \{ \text{ Lemma 4.13 } \} \\ \quad = \mathcal{L}[\![o.x]\!](s^*, h)(s', h') & \{ \text{ Eq. (4.8) } \} \\ \quad = h(s^*(o))(C)(x) & \{ \text{ def. } \mathcal{L}[\![\_]\!] \ \} \\ \quad = h(\alpha)(C)(x) & \{ \text{ def. } s^* \ \} \\ \quad = h(v_1)(D)(x) \ . & \{ \text{ prev. result } \} \end{array}$$

Hence the first condition of the derivation is met.

Observe that Eq. (4.8) implies that

$$(s^*, h)(s', h') \models (\mathsf{defined}(e) \wedge \neg(e = \mathsf{null}))[z/o.x] \ .$$

Then $(s^*, h[\alpha.x_D \mapsto \beta])(s', h') \models \mathsf{defined}(e) \wedge \neg(e = \mathsf{null})$ follows from Lemma 4.13. Hence $\mathcal{E}[\![e]\!](s^*, h[\alpha.x_D \mapsto \beta]) = v_1 \notin \{\bot, null\}$. The third requirement $v_2 \neq \bot$ follows from $(s^*, h)(s', h') \models \mathsf{defined}(e')[z/o.x]$ along the same lines. Hence the computation in Eq. 4.9 can be derived.

Now observe that Eq. 4.8 implies that $(s^*, h)(s', h') \models Q[z/o.x]$. By Lemma 4.13 we then get $(s^*, h[\alpha.x_D \mapsto \beta])(s', h') \models Q$. Since o and z do not occur in Q we also have $(s, h[\alpha.x_D \mapsto \beta])(s', h') \models Q$.

Finally, observe that a computation never depends on the values of logical variables. The existence of the computation in Eq. 4.9 therefore implies the existence of the computation

$$\langle e.x := e', (s, h[\alpha.x_C \mapsto \beta])\rangle \to (s, h)$$

Recall that $\models \{Q\}e.x := e'\{Q'\}$ and $(s, h[\alpha.x_D \mapsto \beta])(s', h') \models Q$. The above computation then implies $(s, h)(s', h') \models Q'$ according to Def. 3.2. □

4.4 Related Work

The weakest precondition calculus for assignments in object-oriented programs described in this chapter has its roots in work by America and De Boer [dB91, AdB94, dB99] on Hoare logics for object-oriented programming languages. However, they never considered languages with subtype polymorphism and inheritance. Consequently, it was hitherto unknown if their approach could be extended to reason about object-oriented languages like Java and C#.

Reus et al. used the above mentioned approach to reason about Java realizations of UML models with OCL annotations [RWH01]. Their Hoare calculus ignores field shadowing, and puts more restrictions on field assignments.

Our weakest precondition calculus for assignments [PdB03b, PdB05b] fully handles field shadowing, and the proposed weakest preconditions require no ad hoc extensions of the assertion language. The field shadowing problem can also be tackled by a preprocessing step that disambiguates fields by annotating them with the classes in which they are declared (see, e.g., [Mül02]). However, field identifiers in programming language expressions lack this kind of annotation. Consequently, one must devise ad hoc syntax for this approach.

Luckham and Suzuki [LS79] extended the syntax of their assertion language with reference classes in order to reason about pointer operations in Pascal. A reference class is a possibly unbounded set of values of data structures like arrays and records that pointers may reference. Their proof rule for pointer assignments substitutes the reference class of the pointer type in the postcondition to model the effect of the assignment.

Poetzsch-Heffter and Müller substitute the reference classes by a global object store in their Hoare rules for assignments in object-oriented programming [PH97, PHM99, Mül02]. Their weakest precondition calculus for assignments replaces references to the object store in the postcondition by references to an updated object store. Both the reference classes as well as the object store are not present in ordinary program expressions.

Cartwright and Oppen studied aliasing in the context of a language with pointers and array assignments [CO81]. Their assertion language also contains additional predicates that model array updates. They do not consider records, which ensures that pointer assignments can essentially still be handled using ordinary substitution operations.

The use of a powerful specification language like higher order logic makes it possible to define weakest preconditions of statements directly in terms of the semantics of statements. The weakest precondition calculus of Jacobs [Jac04] for a subset of Java illustrates this possibility.

Chapter 5

Reasoning about Method Calls

In this chapter, we will develop an approach to reasoning about method calls that can be exploited to verify proof outlines of object-oriented programs. It will turn out that the standard approach to reasoning about method calls in Hoare logics is incompatible with a proof outline logic. We will solve this problem by constructing an adaptation rule that checks whether the specification of a method call follows from the specification of the corresponding method implementation.

However, before we discuss these issues we will first explain why reasoning about method calls in object-oriented programming languages is more challenging than reasoning about procedure calls in ordinary imperative languages. The latter topic has an interesting history of its own in which several rules where proposed that resulted in unsound or incomplete logics [Apt81, Old83].

Reasoning about method calls is more difficult because the state of an object-oriented program is far more complex than the simple mappings from a fixed set of variables to values that are usually considered in classical papers about Hoare rules for procedure calls. The state of an object-oriented program additionally comprises a heap that has no fixed size; an object-oriented program can extend heaps by allocating new objects. This makes it harder to construct an adaptation rule for such languages because an adaptation rule typically replaces all program variables in a formula by logical variables (cf. Section 5.2).

Secondly, we must also find a way to handle dynamic binding. Dynamic binding destroys the ordinary statically determinable connection between a method call and the executed method implementation (cf. Section 2.1). It will no longer be possible, in general, to determine which implementation will be executed as a result of a particular method call. Consequently, we may have to consider several implementations in order to prove the specification of one particular method call.

This chapter is divided into two parts. The first part follows the standard approach to reasoning about procedure calls in Hoare logics. We will develop new versions of the standard Hoare rules which will enable us to reason about method calls in object-oriented programs. However, we will also explain why these rules are not suitable for a proof outline logic. The second part of this chapter, which starts in Section 5.2, focusses on adaptation rules. We will construct an object-oriented adaptation rule that is equally powerful on its own as the set of Hoare rules in the first part of this chapter. Moreover, it has the advantage that it can be employed in proof outlines. Related work is discussed in the last section of this chapter.

5.1 Standard Hoare Rules for Method Calls

In this section we will first summarize the standard approach to reasoning about procedure calls in Hoare logics. The standard approach is also described in, for example, the survey article by Apt [Apt81]. The summary below serves as a gentle introduction to the rules for reasoning about method calls in object-oriented programs in the remainder of this section.

5.1.1 Hoare Rules for Procedure Calls

We will assume in this section that a procedure declaration has the form

$$p \Leftarrow S \ .$$

This syntax declares a procedure p with body S (a statement). A call to procedure p is denoted by $\mathsf{call}\ p()$. For now, we only consider parameterless procedures in a language with global variables.

Hoare proposed rules for reasoning about calls to recursive procedures with several kinds of parameters mechanisms in his seminal paper on procedures and parameters [Hoa71]. He proposed the following rule (in our notation) for reasoning about recursive procedures without parameters:

$$\frac{\{P\}\ \mathsf{call}\ p()\ \{Q\} \vdash \{P\}\ S\ \{Q\}}{\{P\}\ \mathsf{call}\ p()\ \{Q\}} \ . \tag{5.1}$$

Here P and Q are arbitrary first-order formulas over program variables. The rule says that it is valid to conclude $\{P\}\ \mathsf{call}\ p()\ \{Q\}$ if there is a derivation of $\{P\}\ S\ \{Q\}$ that uses the assumption that any call to procedure p already satisfies the desired specification. [1]

This simple rule on its own is, unfortunately, not sufficient. It cannot, for example, handle local variables. We can illustrate this deficiency by means of the procedure declaration $p \Leftarrow \mathsf{skip}$ and the Hoare triple $\{u = 1\}\ \mathsf{call}\ p()\ \{u = 1\}$,

[1] De Bakker [dB80] and Apt [Apt81] both remark that Hoare's rule is a specific instance of the more general rule that is known as Scott's induction rule (see, e.g., [dB80, p. 173]).

where u is a local variable in the context of the call. If we try to derive this Hoare triple using rule (5.1) we are forced to prove that

$$\{u = 1\}\ \mathsf{call}\ p()\ \{u = 1\} \vdash \{u = 1\}\ \mathsf{skip}\ \{u = 1\}\ .$$

In other words, we must prove that the Hoare triple $\{u = 1\}$ skip $\{u = 1\}$ follows from the assumption that the specification of the call satisfies the same specification. This assumption is not necessary here because the given procedure is not recursive. The problem here is the Hoare triple $\{u = 1\}$ skip $\{u = 1\}$. This Hoare triple cannot, in general, be permitted because we cannot allow the local variables that occur in the context of a call to appear in the specification of the body of the corresponding procedure. Consider what could happen if that procedure would also have a local variable u of its own...

The local variable problem can be solved by adding another rule and an additional axiom to our logical system. The required additions are the invariance axiom[2] (5.2) and the conjunction rule (5.3) below.

$$\{P\}\ \mathsf{call}\ p()\ \{P\},\ \text{where}\ \mathsf{vars}(P) \cap \mathsf{vars}(S) = \emptyset \tag{5.2}$$

$$\frac{\{P\}\ \mathsf{call}\ p()\ \{Q\} \quad \{P'\}\ \mathsf{call}\ p()\ \{Q'\}}{\{P \wedge P'\}\ \mathsf{call}\ p()\ \{Q \wedge Q'\}} \tag{5.3}$$

We assume here that $\mathsf{vars}(P)$ denotes the set of all free *global* variables in P, and that $\mathsf{vars}(S)$ denotes the set of global variables that are modified by S. Note that $\{u = 1\}$ call $p()$ $\{u = 1\}$ is a valid instance of the invariance axiom because u is a local variable. Hence this axiom suffices to prove the example Hoare triple above. The conjunction rule can be used to combine instances of the invariance axiom with Hoare triples regarding a call that have been derived using Hoare's recursion rule.

However, there is another problem that will force us to adopt even more rules. In turns out that we also need a way to adapt the specification of a method to what is needed in the context of a call. We will illustrate this point with a simplified version of an example that Apt [Apt81] used to explain the same problem. Consider the following procedure declaration.

$$p \Leftarrow \mathsf{if}\ x = 0\ \mathsf{then\ skip\ else}\ x := x - 1;\ \mathsf{call}\ p();\ x := x + 1\ \mathsf{fi}\ ,$$

Now suppose that we want to prove the Hoare triple $\{x = z\}$ call $p()$ $\{x = z\}$, where x is a global program variable and z is a logical variable. In other words, we want to prove that procedure p restores the initial value of x. The desired property can be derived using rule (5.1) if we can first prove that

$$\{x = z\}\ \mathsf{if}\ x = 0\ \mathsf{then\ skip\ else}\ x := x - 1;\ \mathsf{call}\ p()\ x := x + 1\ \mathsf{fi}\ \{x = z\}\ , \tag{5.4}$$

[2]Note that the invariance axiom is not valid in a proof system for total correctness. Apt has [Apt81] proposed a separate set of rules for total correctness. However, America and De Boer [AdB90] have shown that his logic for total correctness is unsound.

follows from the assumption $\{x = z\}$ call $p()$ $\{x = z\}$. In order to derive (5.4) we must prove that the recursive call in (5.4) satisfies

$$\{x = z - 1\} \text{ call } p() \ \{x = z - 1\} \ . \tag{5.5}$$

We must prove this Hoare triple using the assumption $\{x = z\}$ call $p()$ $\{x = z\}$ because we run into an infinite regression if we try to prove (5.5) by again applying rule (5.1). Intuitively, (5.5) follows from our assumption because the logical variable z is a placeholder for an arbitrary value. However, the assumption $\{x = z\}$ call $p()$ $\{x = z\}$ does not exactly match the desired Hoare triple (5.5).

In this sort of situation, we would like to have additional rules that enable us to adapt a given specification of a call to some other desired specification. The following two substitution rules support this kind of specification adaptation.

$$\frac{\{P\} \text{ call } p() \ \{Q\}}{\{P^z_{z'}\} \text{ call } p() \ \{Q^z_{z'}\}} \tag{5.6}$$

$$\frac{\{P\} \text{ call } p() \ \{Q\} \quad z \notin \mathsf{vars}(Q)}{\{P^z_e\} \text{ call } p() \ \{Q\}} \tag{5.7}$$

Recall from Chapter 4 that P^x_e denotes the formula that is obtained from P by replacing every occurrence of x by expression e. The first substitution rule allows one to replace the free occurrences of a logical variable z by some other logical variable z'. The substitution operation must be capture-avoiding. The second substitution rule enables one to replace a logical variable z in the precondition by a program expression whenever the logical variable does not occur in the postcondition.

We can now derive $\{x = z - 1\}$ call $p()$ $\{x = z - 1\}$ from the assumption $\{x = z\}$ call $p()$ $\{x = z\}$ as follows. Using the first substitution rule we get

$$\{x = z'\} \text{ call } p() \ \{x = z'\} \ . \tag{5.8}$$

Note that $\{z' = z - 1\}$ call $p()$ $\{z' = z - 1\}$ is a valid instance of the invariance axiom. Hence we can derive

$$\{x = z' \wedge z' = z - 1\} \text{ call } p() \ \{x = z' \wedge z' = z - 1\} \ .$$

using the conjunction rule. Observe that $x = z' \wedge z' = z - 1 \rightarrow x = z - 1$ is valid. Hence by the ordinary rule of consequence

$$\{x = z' \wedge z' = z - 1\} \text{ call } p() \ \{x = z - 1\} \ .$$

At this point we use the second substitution rule with the substitution x for z' to obtain

$$\{x = x \wedge x = z - 1\} \text{ call } p() \ \{x = z - 1\} \ .$$

Now $\{x = z-1\}$ call $p()$ $\{x = z-1\}$ can be derived using the rule of consequence.

The rules that we discussed in this section lead to a proof system that is relatively complete [Gor75, Apt81]. We will translate these rules for reasoning about procedure calls into rules for reasoning about method calls in cOOre programs in the following subsection. Subsequently, we will ask ourselves the question whether these rules can be used in the context of proof outlines.

5.1.2 Hoare Rules for Method Calls

We started this chapter with an enumeration of some of the differences between reasoning about procedure calls and reasoning about method calls in object-oriented programs. However, it turns out that the standard rules for reasoning about method calls can nevertheless be adapted in a relatively straightforward way for reasoning about methods calls. We will present the object-oriented counterparts of these rules in this section.

We first give a rule that enables us to prove method specifications. It is the object-oriented counterpart of Hoare's recursion rule in (5.1). Let

$$t\ m(p_1, \ldots, p_n)\ \{\ S\ \mathsf{return}\ e\ \}$$

be the implementation of method m in class C. The following rule suffices for simple recursive methods.

$$\frac{\{P\}\ m@C\ \{Q\} \vdash \{P\}\ S\ \{\mathsf{defined}(e) \rightarrow Q[e/\mathsf{result}]\}}{\{P\}\ m@C\ \{Q\}} \tag{5.9}$$

The rule says that a method specification $\{P\}m@C\{Q\}$ is valid if one is able to prove that the desired specification holds for the body of the method under the assumption that the specification holds for every call to that particular method in the body S. We have defined the meaning of method specifications in Section 3.2. The precondition P in a method specification may refer to the receiver this of the method, and the parameters $p_1, \ldots, p_n$. Local variables are not allowed in P. The postcondition Q may not depend on any local variable or formal parameter. Moreover, we assume in this section that the receiver this does not occur in the postcondition to avoid confusion with occurrences of this in the context of a call.

Recall from Section 3.2 that the special-purpose logical variable result denotes the result value in the final state of the method execution. The statement $\mathsf{return}\ e$ is treated as a (virtual) assignment $\mathsf{result} := e$. Note that we have proved in Section 4.1 that the assertion $\mathsf{defined}(e) \rightarrow Q[e/u]$ is the weakest precondition of an assignment $u := e$ with respect to an postcondition Q. Similarly, we have that $\mathsf{defined}(e) \rightarrow Q[e/\mathsf{result}]$ is the weakest precondition of the assignment $\mathsf{result} := e$ with respect to a postcondition Q.

The rule in (5.9) enables us to derive method specifications. Our next step will be to present a rule that uses such method specifications to infer specifications for a particular call. Recall that a call to a method that returns a result value has the form $u := e_0.m(e_1, \ldots, e_n)$. For simplicity, we will only consider calls for which we can statically determine to which method implementation the call is bound in this section. This situation arises, for example, if m is a private method, or if the implementation of m is not overridden in a subclass. Note that cOOre has no access modifiers, so only the second case may occur in cOOre programs.

Let C be the static type of $[e_0]$, and let the implementation of method m in class C' be the implementation that is either inherited by or declared in class C.

In the latter case, we have $C = C'$. The value of e_0, i.e., the receiver of the call, is an object of (a subclass of) class C. Hence e_0 inherits this implementation if the method is not overridden. Consequently, we know that the implementation in class C' is the implementation that is executed as a result of the call. Let $p_1, \ldots, p_n$ be the formal parameters of this implementation. We then have the following basic rule for method invocations.

$$\frac{\begin{array}{c}\{P\}m@C'\{Q\} \\ Q' \rightarrow (P[e_0, e_1, \ldots, e_n/\mathsf{this}, p_1, \ldots, p_n][u_1, \ldots, u_k/z_1, \ldots, z_k] \\ Q[u_1, \ldots, u_k/z_1, \ldots, z_k] \rightarrow Q''[\mathsf{result}/u]\end{array}}{\{Q'\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q''\}} \tag{5.10}$$

The first assumption of this rule is the specification of the method to which the call is bound. The next two assumptions are two implications that check whether the specification of the call follows from the method specification. The second assumption of the rule verifies whether the precondition of the call implies the precondition of the method invocation. The final assumption checks if the postcondition of the implementation implies the postcondition of the call.

In the implications we find three types of substitution operations. The simultaneous substitution $[e_0, e_1, \ldots, e_n/\mathsf{this}, p_1, \ldots, p_n]$ reflects the context switch, which consists of the assignment of the value of e_0 to this, and the values of the actual parameters to the formal parameters. It is the operation that simultaneously executes the type-preserving substitutions $[e_0/\mathsf{this}]$ and $[e_i/p_i]$, for $i \in \{1 \ldots n\}$ as defined in Section 4.1. Note that the typing rules of cOOre ensure that $[e_i] \preceq [p_i]$ for every actual parameter e_i and every corresponding formal parameter p_i. The substitution $[\mathsf{result}/u]$ that is applied to the postcondition Q'' models a (virtual) assignment of the result value to the local variable u of the caller.

The final substitution operation $[u_1, \ldots, u_k/z_1, \ldots, z_k]$ denotes the simultaneous version of the type-preserving substitutions $[u_1/z_1]$, $\ldots$, $[u_k/z_k]$. It allows us to replace logical variables in the method specification by local variables. The logical variables $z_1, \ldots, z_k$ must be distinct. The final substitution has the same function as the invariance axiom in the previous section. It enables us to prove that the local variables of the caller are not changed during a method call. Consider, for example, the specification $\{z = z'\}\ m@C\ \{z = z'\}$, which is valid for every method because the logical variables z and z' cannot be modified in a program. By choosing the substitution $[u'/z']$ we can deriving $\{z = u'\}\ u := e_0.m(e_1, \ldots, e_n)\ \{z = u'\}$ using rule (5.10) provided that the call is bound to the implementation of method m in class C.

We have shown elsewhere [PdB03b, PdB03c] how these rules for reasoning about method calls can be refined in order to handle dynamic binding. Moreover, we have also shown that the resulting rules can be combined with the substitution rules and the conjunction rule to obtain a Hoare logic for object-oriented programs that is both sound and relatively complete [PdB03c].

5.1.3 Hoare Rules in a Proof Outline Logic

The aim of this thesis is to develop a proof outline logic for object-oriented programs. That is, we are searching for a logic that describes the verification conditions that ensure that an annotated program constitutes a valid proof outline. In the context of method calls, this means that we are looking for a set of verification conditions $Q_1, \ldots, Q_n$ that imply that a particular Hoare triple $\{Q'\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q''\}$ is valid provided that the corresponding method specification $\{P\}m@C'\{Q\}$ is also valid.

The two implications in the assumptions of rule (5.10) seem to be suitable candidates. However, we have explained in Section 5.1 that the Hoare rule in (5.1) only leads to a complete proof system if it is supplemented by two substitution rules, the conjunction rule and the invariance axiom. We must therefore fear that the resulting logic is incomplete if we cannot find a way to integrate these additional rules in either rule (5.9) or rule (5.10).

Our rules are slightly more advanced than Hoare's original recursion rule due to the simultaneous substitution $[u_1, \ldots, u_k/z_1, \ldots, z_k]$ that we allow in our basic method invocation rule (5.10). This substitution integrates the invariance axiom in Hoare's recursion rule. However, a disadvantage of this approach is that we must additionally specify this substitution for every method call in our program annotation. Another drawback is that it forces us to represent the local variables by means of logical variables in the specification of the method that is being called. We have shown in the previous section that we can derive $\{z = u'\}\ u := e_0.m(e_1, \ldots, e_n)\ \{z = u'\}$ from the specification $\{z = z'\}\ m@C\ \{z = z'\}$, but we cannot derive it, for example, from the method specification $\{\mathsf{true}\}\ m@C\ \{\mathsf{true}\}$. The adaptation rule that we will introduce in the following section overcomes these two shortcomings of our basic invocation rule.

What about the two substitution rules? We used these two rules in Section 5.1 to adapt the method specification to what is needed for a particular call. So far, we have not integrated these rules into our invocation rule. It is most likely, therefore, that our present rule inevitably leads to an incomplete proof system.

There is another problem with rule (5.10) too. It is not able to handle expressions of the form $\mathsf{old}(e)$ in the postcondition of a method invocation properly. Suppose that we have a method m in class C with the following specification.

$$\{\mathsf{this} = z\}m@C\{z.x = \mathsf{old}(z.x) + 1\}$$

Apparently, this method increases the value of the x field of its receiver by one. Now consider what happens if we try to verify the following Hoare triple.

$$\{u.x = \mathsf{old}(u.x) + 1\}\ u.m()\ \{u.x = \mathsf{old}(u.x) + 2\}$$

If this method call is bound to the method invocation above, we must prove the following two verification conditions according to rule (5.10).

$$u.x = \mathsf{old}(u.x) + 1 \rightarrow u = z \tag{5.11}$$

$$z.x = \mathsf{old}(z.x) + 1 \rightarrow u.x = \mathsf{old}(u.x) + 2 \tag{5.12}$$

These verification conditions are clearly not valid. We can improve the situation by additionally applying the substitution $[u/z]$, which the rule allows us to do. This results in the following two verification conditions.

$$u.x = \mathsf{old}(u.x) + 1 \rightarrow u = u \tag{5.13}$$

$$u.x = \mathsf{old}(u.x) + 1 \rightarrow u.x = \mathsf{old}(u.x) + 2 \tag{5.14}$$

The first verification condition now becomes valid, but the second verification condition remains invalid. This is caused by the two different interpretations of the $\mathsf{old}(.)$ predicate in the example. The predicate $\mathsf{old}(z.x)$ in the method specification corresponds to the value of $u.x$ prior to the method call, whereas the expression $\mathsf{old}(u.x)$ in the specification of the method call denotes the value of field x of object u in the initial state of the method in whose body the call occurs.

So it turns out that our rule (5.10) is only sound if we do not allow expressions of the form $\mathsf{old}(.)$ in the postcondition Q of the method specification. It is somewhat surprising that a well-known specification technique like the use of these $\mathsf{old}(.)$ predicate, which is also used in, e.g., JML [LBR04], is incompatible with the standard approach to reasoning about method calls in Hoare logic.

Our conclusion is that the standard Hoare rules for procedure calls can be used for reasoning about method calls in object-oriented languages, but there is no obvious way to integrate them in a proof outline logic. We will therefore explore an alternative approach to reasoning about method calls in the following section. We will show that this approach overcomes all the shortcomings of the standard approach that we signalled in this section.

5.2 Adaptation Rules for Method Calls

Hoare discussed the local variable issue that we have described in Section 5.1.1 in his seminal paper on procedures and parameters [Hoa71]. He writes: "What is required is a rather more powerful rule which permits the assumed properties of a recursive call to be adapted to the particular circumstances of that call"[Hoa71, p. 110]. With these words he introduces the first adaptation rule. The aim of this section is to develop an object-oriented adaptation rule. However, we will first have a look at some existing adaptation rules.

5.2.1 Adaptation Rules for Procedure Calls

We will not repeat Hoare's adaptation rule here because Olderog has shown that the rule that Hoare proposed is not the strongest possible rule [Old83]. We will therefore discuss Olderog's stronger adaptation rule in this section.

The starting point of an adaptation rule is a given specification for a call, and some other desired postcondition. The adaptation rule then describes the weakest precondition that the call should satisfy given the particular call specification. If we also have a desired precondition, then we must simply check if that

precondition implies the weakest precondition. This latter situation corresponds to the following version of Olderog's adaptation rule.

$$\frac{\{P\}\ \mathsf{call}\ p()\ \{Q\} \qquad P' \rightarrow ((\forall \bar{z} \bullet P \rightarrow Q[\bar{y}/\bar{x}]) \rightarrow Q'[\bar{y}/\bar{x}])}{\{P'\}\ \mathsf{call}\ p()\ \{Q'\}} \tag{5.15}$$

The conclusion of this rule is the desired call specification. Its first premise is the available (valid) call specification; its second premise is the implication that checks if the desired precondition P' implies the weakest possible precondition.

In this rule, P, P', Q, and Q' are assertions, and $\bar{x}$ is a list of the variables that are modified by procedure p. The list $\bar{y}$ is a list of *fresh* logical variables of the same length as $\bar{x}$. We substitute the program variables $\bar{x}$ by the logical variables $\bar{y}$ in the postconditions Q and Q' in order to distinguish between the initial and the final values of the program variables. This is necessary because the implication brings together preconditions and postconditions — which describe different states — into one formula. The list $\bar{z}$ contains all logical variables that occur free in P and Q. By quantifying over the logical variables in the original call specification the rule reflects the fact that those logical variables are merely placeholders for arbitrary values. Thus it allows us to adapt call specifications.

The soundness of this rule can be explained (informally) as follows. Suppose that we have a computation of the procedure that starts in a state that satisfies P'. Now take the initial state of the computation and assign to the logical variables $\bar{y}$ the values of the program variables $\bar{x}$ in the *final* state of the computation. Thus we have that Q' holds in the final state of the computation if and only if $Q'[\bar{y}/\bar{x}]$ holds in the modified initial state.

Now consider the second premise of the rule. This implication also holds in the modified initial state. Moreover, P' also holds in the modified initial state because the logical variables $\bar{y}$ do not occur in P'. Hence the weakest precondition $(\forall \bar{z} \bullet P \rightarrow Q[\bar{y}/\bar{x}]) \rightarrow Q'[\bar{y}/\bar{x}]$ holds in the modified initial state. We will show that the antecedent of this implication holds in the modified initial state in order to prove its conclusion. This would suffice to prove the soundness of the rule due to the correspondence between Q and $Q'[\bar{y}/\bar{x}]$ that we have indicate above.

To prove the antecedent of the weakest precondition we consider another variant of the modified initial state that is obtained by assigning arbitrary values to the logical variables in $\bar{z}$. Assume that P holds in this third state. We also have a computation that starts in this state because computations do not depend on the values of logical variables. Note that Q must hold in the final state of this computation due to the first premise of the rule. Moreover, the logical variables in $\bar{y}$ will have the same values as the program variables $\bar{x}$ in the final state (the variables in $\bar{z}$ are disjoint from those in $\bar{y}$). Consequently, $Q[\bar{y}/\bar{x}]$ holds in the final state. Then $Q[\bar{y}/\bar{x}]$ holds also in the modified initial state because it does not depend on any of the program variables $\bar{x}$ that may have been modified by the procedure. This proves that the antecedent of the implication holds, which

is, as explained above, the last step in our informal explanation of the soundness of this rule.

Example 5.1. *In order to clarify the use of the adaptation rule, let us revisit the example that we discussed earlier in Section 5.1.1, which involved the following procedure declaration.*

$$p \Leftarrow \mathsf{if}\ x = 0\ \mathsf{then\ skip\ else}\ x := x - 1;\ \mathsf{call}\ p();\ x := x + 1\ \mathsf{fi}$$

We will try to prove again that the Hoare triple

$$\{x = z - 1\}\ \mathsf{call}\ p()\ \{x = z - 1\}$$

follows from the given call specification $\{x = z\}\ \mathsf{call}\ p()\ \{x = z\}$*. If we instantiate the adaptation rule (5.15) we get the verification condition*

$$x = z - 1 \rightarrow (\forall z \bullet x = z \rightarrow y = z) \rightarrow y = z - 1\ ,$$

which is clearly valid. Note that we have replaced the global program variable x *in the postconditions by a fresh logical variable* y *because* x *is modified by* p*.*

Olderog's adaptation rule exploits the weakest precondition that ensures that a desired postcondition holds given a particular call specification. It is also possible, however, to express a variant of his adaptation rule that instead involves the strongest postcondition that follows from an arbitrary precondition. It concerns the following rule.

$$\frac{\begin{array}{c}\{P\}\ \mathsf{call}\ p()\ \{Q\} \\ (P'[\bar{y}/\bar{x}] \wedge (\forall \bar{z} \bullet P[\bar{y}/\bar{x}] \rightarrow Q)) \rightarrow Q'\end{array}}{\{P'\}\ \mathsf{call}\ p()\ \{Q'\}} \tag{5.16}$$

The elements of this rule are similar to the elements of the previous adaptation rule. For example, $\bar{x}$ is again a list of the variables that are modified by p, and $\bar{y}$ once more denotes a corresponding list of fresh logical variables. The list $\bar{z}$ once again contains the logical variables in P and Q.

However, there are also some important differences. The formula

$$P'[\bar{y}/\bar{x}] \wedge (\forall \bar{z} \bullet P[\bar{y}/\bar{x}] \rightarrow Q)$$

describes the state *after* the method call, whereas the corresponding formula in (5.15) describes the state *prior* to the method call. Moreover, the second rule replaces the program variables in the preconditions by logical variables instead of in the postconditions. We will argue in the following section that these differences make this rule much more suitable for use in object-oriented programming than the previous rule.

The soundness of rule (5.16) can be argued along the lines of our informal explanation of the weakest precondition rule. However, for this rule one must consider the final state of the computation, and assign the initial values of the program variables $\bar{x}$ to the logical variables $\bar{y}$. We were able to derive rule (5.16) using Olderog's general analysis of adaptation rules [Old83]. We later discovered that it is also stated by Zwiers et al. [ZHL+96].

5.2.2 An Object-Oriented Adaptation Rule

We have examined two adaptation rules in the previous section that are both tailored to a simple language with global variables. We will now develop an object-oriented version of the last rule that will enable us to assign to every annotated method call $\{Q\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q'\}$ a verification condition which checks whether the annotation of the call follows from the annotation of the corresponding method.

But first we will explain why we cannot state a counterpart of Olderog's original rule (5.15) in an object-oriented setting. Recall that this rule exploits the weakest precondition of a method call with respect to some desired postcondition and a given method specification. This precondition describes the state prior to the call. However, the logical variables $\bar{y}$ that it introduces describe the final state after the call. The final state is likely to contain more objects than the initial state because the called method may allocate new objects. The program variables will presumably reference some of these new objects in the final state. This kind of situation cannot be described by a formula that describes the initial state because consistent states only reference existing objects.

The strongest postcondition variant of the adaptation rule that we presented in the previous section is more promising because its verification condition describes the final state after the method call. This causes no problems because the initial values of the program variables still exist in the final state. The objects that existed prior to the call are simply a subset of the objects in the final state. We will therefore develop an object-oriented counterpart of rule (5.16) in this section.

The following parts of this section describe the changes to the rule that are necessary in order to employ it for reasoning about method calls. Note that we will first develop an adaptation rule for reasoning about calls for which we can statically determine to which method implementation the call is bound; the more complex calls that involve dynamic binding are handled in Section 5.2.3. A soundness proof of the final rule is given in Section 5.2.4.

A Heap Model

We have seen that the adaptation rule replaces every program variable that is modified by the method call by a fresh logical variable in the preconditions. We must therefore consider the question which variables in the precondition of a method call in cOOre should be replaced likewise. The rule for method invocations in the operational semantics of cOOre shows that only the heap is modified by a method call (see Section 2.2.3). Hence we will replace every variable that denotes a heap location by a logical variable. All variables whose values are specified by the stack remain unchanged during a method call. That is, the local variables of the caller and the logical variables retain their values.

Since every variable that denotes a heap value will have to be replaced by a logical variable we will have to find logical variables to represent all heap locations. In other words, we have to build a model of the heap using logical

variables. Recall that the heap stores the existing objects and the values of their fields.

We represent the set of existing objects by means of a fresh logical variable $\mathcal{H}$ of type object^*. This sequence is supposed to contain all objects that are allocated in the heap. Each field that is declared in the program is also represented by a logical variable. For each field x of type t that is declared in some class C we introduce a logical variable $\mathcal{H}(x_C)$ of type t^*. We assume that each of these sequences has the same length as $\mathcal{H}$. Thus $\mathcal{H}(x_C)$ can store the value of the x field of each existing object. The value of the x field of an object that occurs at some position i in the sequence $\mathcal{H}$ is supposed to be stored at the same position i in $\mathcal{H}(x_C)$.

In order to use this heap model we must be able to find the position of an object in the sequence $\mathcal{H}$. We use a function symbol $\mathsf{f} \in \mathsf{F}$ for this purpose. This function is supposed to yield the index in the sequence $\mathcal{H}$ of an object if it is applied to an existing object. For this purpose, we extend the sets of logical expressions with expressions of the form $\mathsf{f}(q)$. Every function $f \in \mathsf{F}$ is assumed to be of type $\mathsf{object} \to \mathsf{int}$. Formally, we will treat functions in the same way as logical variables. That is, we will assume that every stack $s \in \mathit{Stacks}$ also assigns a meaning to function symbols. The value (or interpretation) of a function symbol f is a function with domain $\mathit{dom}(\mathsf{object})$ and range $\mathit{dom}(\mathsf{int})$. We have

$$\mathcal{L}[\![\mathsf{f}(q)]\!](s,h)(s',h') = \begin{cases} \bot & \text{if } v = \bot \\ s(\mathsf{f})(v) & \text{otherwise} \end{cases},$$

where $v = \mathcal{L}[\![q]\!](s,h)(s',h')$. The substitutions operations that we defined in the previous chapter for reasoning about assignments are supposed to handle expressions of the form $\mathsf{f}(q)$ in the same way as expressions of the form $\mathsf{op}(q)$, where op is an unary operator on an element of a primitive type.

In order to fully exploit this heap model we need four additional axioms regarding its constituents, which all can be expressed in our assertion language cOOral.

$$(\forall i \bullet 0 \leq i < \mathcal{H}.\mathsf{length} \to \mathsf{f}(\mathcal{H}[i]) = i) \tag{5.17}$$

$$\mathsf{f}(\mathsf{null}) = \mathsf{length}(\mathcal{H}) \tag{5.18}$$

$$\mathsf{length}(\mathcal{H}(x_C)) = \mathsf{length}(\mathcal{H}) \tag{5.19}$$

$$\mathcal{H}(x_C) \sqsubseteq \mathcal{H} :: \mathsf{null} \tag{5.20}$$

The first formula states that the index function yields the index of each object in $\mathcal{H}$. It also implies that each object occurs at most once in the heap. The second formula ensures that $\mathsf{f}(\mathsf{null})$ does not denote a valid index of $\mathcal{H}$ ($\mathsf{length}(\mathcal{H})$ is bigger than the last valid index $\mathsf{length}(\mathcal{H}) - 1$). The third formula expresses the above-mentioned assumption that every sequence $\mathcal{H}(x_C)$ has the same length as $\mathcal{H}$. We write $z \sqsubseteq \mathcal{H} :: \mathsf{null}$ in the last formula as an abbreviation of the formula $(\forall i \bullet 0 \leq i < \mathsf{length}(\mathcal{H}(x_C)) \to \mathcal{H}(x_C)[i] = \mathsf{null} \vee \mathcal{H}(x_C)[i] \in \mathcal{H})$. Thus the last formula, which should hold for every field $x : t$ declared in some class C if t is a reference type, implies that there are no dangling references in the heap.

We will denote the conjunction of (5.17) -(5.20) by heap. Often we will want to say additionally that every variable in a particular set of local and logical variables V references an object in the old heap. We use the predicate heap_V for this purpose. It denotes the formula $\mathsf{heap} \wedge \bigwedge_{v \in V} v \in \mathcal{H}$ (or $v \sqsubseteq \mathcal{H}$ instead of $v \in \mathcal{H}$ if v is a logical variable of some type C^*).

Bounded Quantification

The heap model that we described above enables us to replace expressions that reference heap locations by the corresponding logical variables in the heap model. But before we describe this translation step we first draw attention to the consequences of heap extensions for the adaptation rule.

Methods can not only modify the heap by assigning to fields, but they can also *extend* it by allocating new objects. And object allocations extend quantification domains. Suppose, for example, that the formula

$$(\forall s : \mathit{Singleton} \bullet s = \mathsf{this})$$

holds prior to a method call. This formula states that the present receiver is the only existing instance of class *Singleton* in the current state. This formula will no longer hold after the call if the method creates new instances of class *Singleton*.

Recall that the verification condition of the adaptation rule 5.16 brings together both preconditions and postconditions in one formula. It even enables us to use the precondition of the call to prove the desired postcondition. It would, however, be wrong to use the above mentioned precondition to prove that there also exists only one instance of class *Singleton* after the method call since this property may no longer hold.

This issue can be solved by explicitly limit the quantification domain of quantifiers to the objects that existed before the call. Recall that the sequence $\mathcal{H}$ is assumed to contain the objects that exist in a particular heap. We will therefore restrict quantification over objects of a class C to objects in the sequence $\mathcal{H}$ in assertions that describe the state prior to a call. For this purpose, we introduce a bounded form of quantification.

Let $\mathcal{H}$ be a logical variable of type object^*. Then we define the bounded variant $(\exists z \in \mathcal{H} : t \bullet P)$ of an expression $(\exists z : t \bullet P)$ as follows.

$$\begin{array}{rcl}
(\exists z \in \mathcal{H} : t \bullet P) & = & (\exists z : t \bullet P) \text{ for } t \in \{\mathsf{int}, \mathsf{boolean}\} \\
(\exists z \in \mathcal{H} : t^* \bullet P) & = & (\exists z : t^* \bullet P) \text{ for } t \in \{\mathsf{int}, \mathsf{boolean}\} \\
(\exists z \in \mathcal{H} : C \bullet P) & = & (\exists z : C \bullet (z = \mathsf{null} \vee z \in \mathcal{H}) \wedge P) \\
(\exists z \in \mathcal{H} : C^* \bullet P) & = & (\exists z : C^* \bullet z \sqsubseteq \mathcal{H} :: \mathsf{null} \wedge P)
\end{array}$$

Note that formulas that quantify over primitive types retain their original meaning; we introduce bounded variants of quantification over such types merely for notational convenience.

$$
\begin{array}{rcl}
\lfloor \mathsf{null} \rfloor & = & \mathsf{null} \\
\lfloor \mathsf{this} \rfloor & = & \mathsf{this} \\
\lfloor u \rfloor & = & u \\
\lfloor q.x \rfloor & = & \mathcal{H}(x_C)[\mathsf{f}(\lfloor q \rfloor)], \text{ where } C = \mathsf{origin}([q], x) \\
\lfloor (C)q \rfloor & = & (C)\lfloor q \rfloor \\
\lfloor q \text{ instanceof } C \rfloor & = & \lfloor q \rfloor \text{ instanceof } C \\
\lfloor q_1 \ ? \ q_2 : q_3 \rfloor & = & \lfloor q_1 \rfloor \ ? \ \lfloor q_2 \rfloor : \lfloor q_3 \rfloor \\
\lfloor q = q' \rfloor & = & \lfloor q \rfloor = \lfloor q' \rfloor \\
\lfloor \mathsf{op}(q_1, \ldots, q_n) \rfloor & = & \mathsf{op}(\lfloor q_1 \rfloor, \ldots, \lfloor q_n \rfloor) \\
\lfloor z \rfloor & = & z \\
\lfloor \mathsf{old}(e) \rfloor & = & \mathsf{old}(e) \\
\lfloor z[q] \rfloor & = & z[\lfloor q \rfloor] \\
\lfloor \mathsf{length}(z) \rfloor & = & \mathsf{length}(z) \\
\lfloor \mathsf{undefined} \rfloor & = & \mathsf{undefined} \\
\lfloor \mathsf{defined}(q) \rfloor & = & \mathsf{defined}(\lfloor q \rfloor) \\
\lfloor \neg Q \rfloor & = & \neg \lfloor Q \rfloor \\
\lfloor Q \wedge Q' \rfloor & = & \lfloor Q \rfloor \wedge \lfloor Q' \rfloor \\
\lfloor (\exists z \bullet Q) \rfloor & = & (\exists z \in \mathcal{H} \bullet \lfloor Q \rfloor)
\end{array}
$$

Figure 5.1: The definition of $\lfloor Q \rfloor$.

Building Dual Heap Formulas

The verification condition of an adaptation rule always describes two states: the state prior to the method call and the state after the method call. We have argued above that it suffices to distinguish between the initial heap and the final heap for method calls. The verification condition of our adaptation rule will therefore be a dual heap formula: the ordinary heap references will describe the final heap, and the logical variables of the heap model will describe the initial heap.

We will build dual heap formulas by means of a syntactical operation $\lfloor . \rfloor$. This operation will be an object-oriented counterpart of the substitution $[\overline{y}/\overline{x}]$ in Equation 5.16. The operation achieves the meaning shift by replacing variables that reference heap locations by the corresponding logical variables in the heap model, and by restricting quantification to the objects in the heap model. We define $\lfloor Q \rfloor$ by induction on the structure of Q. All cases of $\lfloor Q \rfloor$ are listed in Figure 5.1.

The most interesting case is $\lfloor q.x \rfloor$ because $q.x$ references a heap location. An expression $q.x$ is replaced by the corresponding variables in the heap model. Let $C = \mathsf{origin}([q], x)$ (field x is defined in class C). Then we have

$$\lfloor q.x \rfloor = \mathcal{H}(x_C)[\mathsf{f}(\lfloor q \rfloor)] \ .$$

Quantification is restricted to the objects in the heap model:

$$\lfloor (\exists z \bullet Q) \rfloor = (\exists z \in \mathcal{H} \bullet \lfloor Q \rfloor) \ .$$

All other cases are straightforward.

The dual heap is a 'real' heap: the initial heap of the call. In the following definition we show how we can reconstruct this heap from the values of the dual heap variables using the $\lfloor . \rfloor$ function. In fact, if (s, h) is the final state of the *caller* after a method call, then $(\mathsf{start}_s, \mathsf{start}_h)$ as defined below is the initial heap of the corresponding execution of the method body.

Definition 5.1. *Let $\mathcal{H}$ be a logical variable of type* object^**, and let $\mathcal{H}(x_C)$ be a logical variable of type t^*, for every field $x : t$ declared in some class C. Let* $\mathsf{f} : \mathsf{object} \rightarrow \mathsf{int}$ *be a function symbol, and let e_0 be an expression such that $[e_0] \sim [\mathsf{this}]$. Furthermore, let $p_1, \ldots, p_n$ be a sequence of formal parameters with a corresponding sequence $e_1, \ldots, e_n$ of actual parameters such that $[e_i] \preceq [p_i]$, for $i \in \{1 \ldots n\}$. Finally, let (s, h) be a state such that $(s, h) \models \mathsf{heap}_V$ and $\mathcal{E}[\![\lfloor e_0 \rfloor]\!](s, h) \in dom([\mathsf{this}])$, where V is the set that contains both* this *and all local variables in e_i, for $i \in \{0 \ldots n\}$. Then $(\mathsf{start}_s, \mathsf{start}_h)$ is the state such that*

- $\mathsf{start}_s(\mathsf{this}) = \mathcal{N}[\![\lfloor e_0 \rfloor]\!](s, h)$, $\mathsf{start}_s(p_i) = \mathcal{N}[\![\lfloor e_i \rfloor]\!](s, h)$ *for $i \in \{1 \ldots n\}$, and* $\mathsf{start}_s(v) = \mathsf{init}([v])$ *for every other local variable or parameter v, and*
- *for every object o,* $\mathsf{start}_h(o)$ *is defined if and only if o occurs in $s(\mathcal{H})$, and moreover, for every field x in class C, and every index i with $0 \leq i < m$ we have* $\mathsf{start}_h(g(i))(C)(x) = g'(i)$ *if $g(i) \in dom(C) \setminus \{\mathsf{null}\}$, where $(g, m) = s(\mathcal{H})$ and $(g', m') = s(\mathcal{H}(x_C))$.*

The clause $(s, h) \models \mathsf{heap}_V$ in Definition 5.1 is necessary to ensure that $(\mathsf{start}_s, \mathsf{start}_h)$ is a valid state (see also Lemma 5.1 below).

Verification Conditions of Method Calls

So far, we have defined several object-oriented counterparts of the main building blocks of the adaptation rule in (5.16). We will now use these parts to build object-oriented adaptation rules. Our eventual goal is an adaptation rule for reasoning about dynamically bound method calls of the form $u := e_0.m(e_1, \ldots, e_n)$. We will present such a rule in Section 5.2.3. In the remainder of this section we will present several adaptation rules for simpler method calls. We will start with very basic method calls in an object-oriented context, and then move step-by-step towards more complex method calls.

The most simple kind of method calls in object-oriented languages are calls to *static* methods. Static method have no receivers, and consequently the keyword this is forbidden in the implementation and specification of a static method. We will assume that each static method has a unique name m, and that $m(e_1, \ldots, e_n)$ denotes a call of method m with actual parameters $e_1, \ldots, e_n$.

The outline of the adaptation rule for this kind of method call is as follows (where 6.18 denotes the verification condition of the call).

$$\frac{\{P\}\ m@C\ \{Q\} \qquad \mathit{VC1}}{\{Q'\}\ m(e_1, \ldots, e_n)\ \{Q''\}} \tag{5.21}$$

We will make the usual assumption that the postcondition Q contains no occurrences of the parameters $p_1, \ldots, p_n$ of the method or of any other local variable. And for simplicity, we will initially also assume that it contains no subexpressions of the form $\mathsf{old}(.)$.

The verification condition *VC1* below has the same structure as its counterpart in rule (5.16).

$$\lfloor \bigwedge_{i=1}^{n} \mathsf{defined}(e_i) \rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q' \rfloor \wedge (\forall \bar{z} \in \mathcal{H} \bullet \lfloor P[\bar{e}/\bar{p}] \rfloor \rightarrow Q) \rightarrow Q'' \quad (VC1)$$

However, it also contains some new elements. The first new element in (*VC1*) is the clause $\bigwedge_{i=1}^{n} \mathsf{defined}(e_i)$, which state that the actual parameters evaluate normally; note that the method call does not terminate if this condition does not hold. Hence we can safely add this assumption. The $\lfloor . \rfloor$ operator is applied to this clause because it describes the situation in the initial heap, which is the dual heap in this verification condition.

The second new element is the predicate heap_V. Here, V is the set that contains this and every local or logical variable of some reference type C or C^* occurring in P, Q or e_i, for $i \in \{0 \ldots n\}$. The additional assumption $\bigwedge_{v \in V} v \in \mathcal{H}$ is valid because the values of these variables are not modified during the method call. Therefore these values must be part of the initial heap.

The formula heap_V ensures that every expression $\lfloor e \rfloor$ denotes an object in the dual heap.

Lemma 5.1. *Let (s, h) be an arbitrary state, and let (s', h') be a compatible freeze state. Then we have, for every program expression e with $[e] \preceq \mathsf{object}$, that $(s, h)(s', h') \models \mathsf{heap}_V$ and $\mathcal{L}[\![\lfloor e \rfloor]\!](s, h)(s', h') \notin \{null, \bot\}$ imply that*

$$(s, h)(s', h') \models 0 \leq \mathsf{f}(\lfloor e \rfloor) < \mathsf{length}(\mathcal{H}) \text{ and } (s, h)(s', h') \models \lfloor e \rfloor \in \mathcal{H} \ ,$$

provided that V contains both this and every local variable in e.

Proof. By structural induction on e. □

Note that in (*VC1*) we restrict the meaning of the call's precondition Q' to the initial heap by means of the $\lfloor . \rfloor$ operator. This operator is also applied to the precondition P of the method. The substitution $[\bar{e}/\bar{p}]$ on P replaces the formal parameters by the actual parameters. We assume that $\bar{e} = e_1, \ldots, e_n$ and $\bar{p} = p_1, \ldots, p_n$. The list $\bar{z}$ again contains all logical variables that occur free in P or Q.

Adding support for result value handling is straightforward. Suppose that we have a call $u := m()$ to a static method m that returns a value. This statement will execute m and subsequently assign the result value to the local variable u. Recall that result denotes the result value in the postcondition Q. Our verification condition must ensure that the postcondition Q'' holds after this assignment, which is equivalent to checking that $Q''[\mathsf{result}/u]$ holds after the method call because $Q''[\mathsf{result}/u]$ is the weakest precondition of the

assignment $u := \mathsf{result}$. Hence we get the following verification condition for the call $u := m()$.

$$\lfloor \bigwedge_{i=1}^{n} \mathsf{defined}(e_i) \rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q' \rfloor \wedge (\forall \bar{z} \in \mathcal{H} \bullet \lfloor P[\bar{e}/\bar{p}] \rfloor \rightarrow Q) \rightarrow Q''[\mathsf{result}/u] \qquad (VC2)$$

The special-purpose logical variable result should not be included in the list $\bar{z}$ because it does not represent an arbitrary value.

In our next step we move to non-static methods. In other words, we will now consider methods that have a specific receiver stored in their implicit this parameter. We will assume that a call to a method of this kind has the form $u := e_0.m(e_1, \ldots, e_n)$, where e_0 denotes the receiver of the method. For now, we will assume that we can statically determine to which method the call will be bound; dynamic binding is discussed in Section 5.2.3. The outline of our new adaptation rule is as follows.

$$\frac{\{P\}\ m@C\ \{Q\} \qquad VC3}{\{Q'\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q''\}} \qquad (5.22)$$

In principle, one can treat the implicit receiver parameter in the same way as ordinary parameters by replacing this in the precondition P of the method by e_0. This yields verification condition $VC3$, where $[e_0, \bar{e}/\mathsf{this}, \bar{p}]$ denotes the simultaneous substitution that replaces this by e_0, and every formal parameter in $\bar{p}$ by its corresponding actual parameter in $\bar{e}$.

$$\lfloor \bigwedge_{i=0}^{n} \mathsf{defined}(e_i) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q' \rfloor \\ \wedge ((\forall \bar{z} \in \mathcal{H} \bullet \lfloor P[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor \rightarrow Q) \rightarrow Q''[\mathsf{result}/u] \quad (VC3)$$

Note that we have also added the clauses $\mathsf{defined}(e_0)$ and $\neg(e_0 = \mathsf{null})$ to the antecedent of the verification condition.

However, we will go one step further. More specifically, we will also add support for occurrences of this in the postcondition Q of the method. This is possible because this is a readonly parameter, which implies that its final value is known in the context of the method call.

So let us assume that the method postcondition Q may contain occurrences of this. This keyword may also occur in the precondition Q' and the postcondition Q'' of the method call. In those contexts, however, it denotes the receiver of the method in whose body the present call occurs, which is not necessarily the same object as the receiver of the call in $u := e_0.m(e_1, \ldots, e_n)$. In our verification condition, we will therefore substitute occurrences of this by a fresh logical variable rec of the same type. Moreover, we add the clause $\mathsf{rec} = \mathsf{this}$ to the precondition of the method in order to make the desired correspondence explicit. The occurrence of this in this clause (and every other occurrence of this in the precondition P) will be replaced by e_0 as a result of the substitution $[e_0, \bar{e}/\mathsf{this}, \bar{p}]$. The proposed changes are embedded in verification condition

VC4.

$$\lfloor \bigwedge_{i=0}^{n} \mathsf{defined}(e_i) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q' \rfloor$$
$$\wedge (\forall \bar{z} \in \mathcal{H} \bullet \lfloor (P \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor \rightarrow (Q[\mathsf{rec}/\mathsf{this}])) \rightarrow Q''[\mathsf{result}/u] \qquad (VC4)$$

The final extension that we will discuss in this section concerns support for the $\mathsf{old}(.)$ construct. An expression $\mathsf{old}(e)$ in the postcondition Q in *VC4* denotes the value of e in the initial state of the call $u := e_0.m(e_1, \ldots, e_n)$. Note that similar expressions may also occur in Q' and Q''. However, an expression $\mathsf{old}(e)$ in Q' or Q'' does *not* denote the value of e in the same initial state; such an expression denotes the value of e in the initial state of the execution of the method in whose body the statement $u := e_0.m()$ occurs. The value of an expression $\mathsf{old}(e)$ of this latter kind is stored in the freeze state, whereas the value of an expression $\mathsf{old}(e)$ of the former kind is determined by the values of the dual heap variables. We have already seen these two different interpretations of $\mathsf{old}(e)$ in Section 5.1.3.

In order to represent this situation correctly in our verification conditions, we must replace every occurrence of an expression of the form $\mathsf{old}(e)$ in the postcondition Q by an expression that denotes the value of e in the initial state. It should be clear that the expression $\lfloor e[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor$ denotes this value. Therefore we will define a syntactical operation that replaces every occurrence of an expression of the form $\mathsf{old}(e)$ in a formula by $\lfloor e[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor$. For brevity, we will assume in the remainder of this thesis that $\mathsf{g}(e)$ and $\mathsf{g}(P)$ abbreviate $\lfloor e[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor$ and $\lfloor P[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor$, respectively.

With function g we can define the above-mentioned substitution operation, which we denote by $[\mathsf{g}(.)/\mathsf{old}(.)]$. It is defined in the same way as $[./\mathsf{old}(.)]$. It only differs from $[./\mathsf{old}(.)]$ in the following case:

$$\mathsf{old}(e)[\mathsf{g}(.)/\mathsf{old}(.)] = \mathsf{g}(e) \ .$$

If we apply this operation to the postcondition of the method in (*VC4*) we get verification condition (*VC5*).

$$\lfloor \bigwedge_{i=0}^{n} \mathsf{defined}(e_i) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q' \rfloor \wedge$$
$$(\forall \bar{z} \in \mathcal{H} \bullet \mathsf{g}(P \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (Q[\mathsf{rec}/\mathsf{this}][\mathsf{g}(.)/\mathsf{old}(.)]))$$
$$\rightarrow Q''[\mathsf{result}/u] \quad (VC5)$$

The following lemma states that function $\mathsf{g}(e)$ indeed denotes the value of e in the initial state of the method execution.

Lemma 5.2. *Let* $(s, h), \mathcal{H}$, $\mathcal{H}(x_C)$, $p_1, \ldots, p_n$, $e_1, \ldots, e_n$, this *and* e_0 *be given as in Definition 5.1. Let* (s', h') *be an arbitrary freeze state that is compatible with* (s, h)*, and let* Q *be an assertion that only contains expressions of the form*

$\mathsf{old}(e)$ *in which every occurrence of a local variable concerns an element of the parameter list* $p_1, \ldots, p_n$*. Then*

$$(s,h)(\mathsf{start}_s, \mathsf{start}_h) \models Q \text{ if and only if } (s,h)(s',h') \models Q[\mathsf{g}(.)/\mathsf{old}(.)] \ ,$$

where g *is the syntactical operation that takes an expression* e *and returns the expression* $\lfloor e[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor$*.*

Proof. By structural induction on Q. The only non-trivial case concerns a logical expression of the form $\mathsf{old}(e)$. For this case, we can prove that, for every expression e in $\mathsf{old}(e)$, we have $\mathcal{E}[\![e]\!](\mathsf{start}_s, \mathsf{start}_h) = \mathcal{N}[\![\mathsf{g}(e)]\!](s,h)$ by structural induction on e. □

Our adaptation rule reveals an interesting practical advantage of the use of $\mathsf{old}(.)$ predicates to refer to initial values over the standard approach that uses logical variables to freeze initial values: it reduces the number of logical variables in $\bar{z}$ in *VC5*. Observe that every logical variable that is used to freeze an initial value in the method specification must be added to $\bar{z}$. This leads to a more complex formula. When proving the verification condition we will have to find out which values of the logical variables in $\bar{z}$ satisfy the precondition of the method invocation. By contrast, the operation $[\mathsf{g}(.)/\mathsf{old}(.)]$ automatically replaces every expression $\mathsf{old}(e)$ by the corresponding value in the initial state without requiring additional logical variables.

Let us examine a small example proof outline with a method call and its resulting verification condition. The example that we give below mainly illustrates the elegant way in which the rule handles local variables.

Example 5.2. *An example proof outline is listed in Fig. 5.2. It concerns a simple class with two methods. The second method calls the first method with the statement* $\mathsf{this}.setX(p)$*. This call has the following verification condition if the setX-method is not overridden in a subclass.*

$$\begin{aligned}\mathsf{defined}(\mathsf{this}) \wedge \mathsf{defined}(p) \wedge \neg(\mathsf{this} = \mathsf{null}) \wedge \mathsf{heap}_V \wedge p = \mathsf{old}(p) \wedge b = \mathsf{old}(\mathsf{this}.x = p) \\ \wedge (\forall \mathsf{rec} : C \bullet \mathsf{rec} \in \mathcal{H} \rightarrow (\mathsf{true} \wedge \mathsf{rec} = \mathsf{this} \rightarrow \mathsf{rec}.x = p)) \\ \rightarrow \mathsf{this}.x = \mathsf{old}(p) \wedge b = \mathsf{old}(\mathsf{this}.x = p)\end{aligned}$$

We did not expand the heap_V*-predicate for brevity; it represents the conjunction of* heap *and the formula* $\mathsf{this} \in \mathcal{H}$ *in this verification condition. The variable* rec *is a logical variable, and is therefore universally bound in the verification condition. Thus it becomes possible to adapt the specification to what is needed in the context of the call. The method specification implies the specification of the call if we choose* this *for* rec*.*

Observe that the clause $b = \mathsf{old}(\mathsf{this}.x = p)$ *in the precondition of the call can be used to prove the same clause in the postcondition. Thus there is no need for a separate invariance axiom to reason about local variables in the specification of the call, and we also do not have to represent the local variables of the caller in the method specification by means of logical variables.*

```
class C {
  int x ;
  requires true;
  ensures this.x = old(p);
  void setX(int p) {
    assert p = old(p);
    this.x := p ;
    assert this.x = old(p);
  }

  requires true;
  ensures this.x = old(p) ∧ result = old(this.x = p);
  boolean testAndSetX(int p) {
    assert p = old(p) ∧ this.x = old(this.x);
    boolean b ;
    assert p = old(p) ∧ this.x = old(this.x);
    b := (this.x = p) ;
    assert p = old(p) ∧ b = old(this.x = p);
    this.setX(p) ;
    assert this.x = old(p) ∧ b = old(this.x = p);
    return b ;
  }
}
```

Figure 5.2: An example proof outline that involves a method call and local variables.

We will present another example in order to motivate our use of bounded quantification in the verification conditions of our adaptation rules.

Example 5.3. *Suppose that we have a class C with a field x : object and a method m with the following specification:*

$$\{\mathsf{true}\}\ m@C\ \{\neg(\mathsf{this}.x = \mathsf{null}) \wedge \neg(\mathsf{this}.x = z)\}$$

Admittedly, this specification is somewhat unusual: it involves a logical variable z which does not occur in the precondition. In this specification, z represents an arbitrary object in the initial state of the method execution. We can use logical variables for this purpose because the initial value and the final value of a logical variable are always the same. Therefore the final value of z is an object that already exists in the initial state. Consequently, this specification says that m allocates at least one new object and that it assigns this object to the x field of its receiver.

Now suppose that we want to prove a similar specification for a call to this method: $\{\mathsf{true}\}\ \mathsf{this}.m()\ \{\neg(\mathsf{this}.x = \mathsf{null}) \wedge \neg(\mathsf{this}.x = z')\}$. *This call gets the*

verification condition

$$\begin{aligned}&\mathsf{defined}(\mathsf{this}) \wedge \neg(\mathsf{this} = \mathsf{null}) \wedge \mathsf{heap}_V \wedge \mathsf{true} \wedge \\ &(\forall \mathsf{rec} \in \mathcal{H} \bullet (\forall z \in \mathcal{H} \bullet (\mathsf{true} \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (\neg(\mathsf{rec}.x = \mathsf{null}) \wedge \neg(\mathsf{rec}.x = z)))) \\ &\qquad \rightarrow \neg(\mathsf{this}.x = \mathsf{null}) \wedge \neg(\mathsf{this}.x = z') \ . \end{aligned} \quad (5.23)$$

The clause $z \in \mathsf{rec}$ *is very important in this verification condition. Without it, the second line of (5.23) becomes a contradiction if we choose* this *for* rec*, and* $\mathsf{this}.x$ *for* z*. For this would result in the clause* $\neg(\mathsf{this}.x = \mathsf{this}.x)$*! Fortunately, this scenario is not possible because we cannot prove that* $\mathsf{this}.x \in \mathcal{H}$*!*

It becomes clear that this verification condition holds if we expand the heap_V *formula. This yields the formula* $\mathsf{heap} \wedge \mathsf{this} \in \mathcal{H} \wedge z' \in \mathcal{H}$*. We need the information regarding* this *and* z' *in this formula because it enables us to instantiate the second line of (5.23) with the corresponding two values. This suffices to prove (5.23).*

5.2.3 Dynamic Binding

The final hurdle in this chapter on reasoning about method calls is dynamic binding. Dynamic binding destroys the static connection between a method call and its implementation: we cannot assume any longer that we know which method implementation will be executed as a result of a particular method call in our code. The dynamic type of the receiver now determines which method implementation is executed.

For the convenience of the reader, we repeat here the rule in the operational semantics that specifies the behavior of method calls.

$$\frac{\begin{array}{c}\mathcal{E}[\![e_0]\!](s,h) = o = (C,i) \\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for } i \in \{1 \ldots n\} \\ \mathcal{E}[\![e]\!](s',h') = v \neq \bot \\ \mathsf{meth}(C,m) \equiv t\ m(p_1,\ldots,p_n)\{\ S\ \mathsf{return}\ e\ \} \\ \langle S, (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_i, \ldots, v_n], h)\rangle \rightarrow (s',h')\end{array}}{\langle u := e_0.m(e_1,\ldots,e_n), (s,h)\rangle \rightarrow (s[u \mapsto v], h')} \quad (MC_1)$$

The rule reveals that it is necessary to evaluate the expression e_0 that denotes the receiver of the call to determine to which method implementation the call is bound. Recall that $\mathsf{meth}(C,m)$ denotes the implementation of method m that is either declared in class C or otherwise inherited by C from its superclass.

The dynamic type of the receiver is always a subtype of the static type of e_0. This fact is expressed by the lemma below.

Lemma 5.3. *For every well-typed expression* e *with* $[e] = C$*, and every state* (s,h)*, we have that* $\mathcal{E}[\![e]\!](s,h) = (D,i)$ *implies* $D \preceq C$*.*

Proof. From Lemma 3.2 follows $\mathcal{E}[\![e]\!](s,h) = \mathcal{L}[\![e]\!](s,h)(s',h')$ for every (s',h') that is compatible with (s,h). Because $(D,i) \neq \bot$ we have $\mathcal{E}[\![q]\!](s,h) \in dom(C)$

by Lemma 3.1. Since $\mathcal{E}[\![e]\!](s,h)$ is also different from the null reference we get $D \preceq C$ from Lemma 2.2. □

Because we cannot statically determine the dynamic type of the receiver, we will have to consider all implementations of method m that are declared in or inherited by subclasses of the static type of the receiver. Each of these implementations may be bound to the call, and each specification of one of these implementations must support the desired specification of the call. It seems, therefore, that our proof outline logic will have to generate a verification condition for each subclass of the static type of the receiver.

Fortunately, we can often reduce the number of verification conditions. We will show that it suffices to generate a verification condition for each method implementation that belongs to some subclass of the static type of the receiver. This reduces the number of verification conditions because not every subclass will have its own implementation; a class may also reuse the implementation that it inherits.

This approach can only be used if every subclass of the static type of the receiver is available for inspection. In other words, our solution cannot be used in the context of open, extensible programs without rechecking part of the proofs. We discuss an alternative solution that is based on behavioral subtyping [Ame91, LW94] in Chapter 8.

Suppose again that we consider a call $u := e_0.m(e_1,\ldots,e_n)$. Let $\mathsf{prov}(C,m)$ denote the class in which the implementation of method m for objects of class C is declared. That is, $\mathsf{prov}(C,m)$ is class C if C declares a method m, and otherwise $\mathsf{prov}(C,m)$ is equal to $\mathsf{prov}(D,m)$, where D is the direct superclass of class C. By $\mathsf{impls}([e_0],m)$ we denote the set of classes that provide an implementation that may be bound to the call. Formally, we have

$$\mathsf{impls}(C,m) = \{D \mid \mathsf{prov}(E,m) = D \text{ for some subtype } E \text{ of } C\} \ .$$

The outline of the rule for dynamically bound method invocations is then as follows.

$$\frac{\begin{array}{c}\mathsf{impls}([e_0],m) = \{C_1,\ldots,C_k\} \\ \{P_1\}\ m@C_1\ \{Q_1\},\ldots,\{P_k\}\ m@C_k\ \{Q_k\} \\ VC_1,\ldots,VC_k\end{array}}{\{Q\}\ u := e_0.m(e_1,\ldots,e_n)\ \{Q'\}} \tag{5.24}$$

Here VC_j, for $j \in \{1\ldots k\}$, denotes the verification condition that corresponds to the implementation in class C_j. It is the following formula.

$$\begin{array}{r}\lfloor \mathsf{boundto}(e_0,C_j,m)\rfloor \wedge \lfloor(\bigwedge_{i=0}^{n} \mathsf{defined}(e_i))\rfloor \wedge \lfloor\neg(e_0 = \mathsf{null})\rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q\rfloor \wedge \\ (\forall \bar{z} \in \mathcal{H} \bullet \mathsf{g}_j(P_j \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (Q_j[\mathsf{rec}/\mathsf{this}][\mathsf{g}_j(.)/\mathsf{old}(.)])) \\ \rightarrow Q'[\mathsf{result}/u] \quad (VC_j)\end{array}$$

This formula is almost identical to $(VC5)$, and its elements are also defined in the same way as their counterparts in $(VC5)$. We assume that $\bar{p}^j = p_1^j,\ldots,p_n^j$

are the formal parameters of the implementation of method m in class C_j, and that $[\mathsf{rec}] = C_j$ in (VC_j). The function g_j is the syntactical function that takes a formula P and returns the formula $\lfloor P[e_0, \bar{e}/\mathsf{this}, \bar{p}^j] \rfloor$.

The only new element in (VC_j) is the formula $\mathsf{boundto}(e_0, C_j, m)$, which is a formula that implies that e_0 is an instance of a class that inherits the implementation of method m in class C_j. A class inherits the implementation in class C if it is a subclass of class C, and if it is not a subclass of some other class that overrides the implementation in class C. We denote the set of classes that override the implementation of method m in class C by $\mathsf{overrides}(C)(m)$. We have $D \in \mathsf{overrides}(C)(m)$ if (1) class D is a proper subclass of class C, and (2) class D provides an implementation of method m, and moreover, (3) class C has no other proper subclass E such that D is a proper subclass of E, and E also provides an implementation of method m.

Then $\mathsf{boundto}(e_0, C_j, m)$ is the formula

$$e_0 \ \mathsf{instanceof}\ C_j \wedge \bigwedge_{D \in \mathsf{overrides}(C_j)(m)} \neg(e_0 \ \mathsf{instanceof}\ D) \ ,$$

which says that the receiver e_0 of the method call is an instance of some some subclass of class C_j, and it is not a subclass of some class that overrides the implementation in class C_j. This assumption is valid because the call will not be bound to the implementation that corresponds to (VC_j) if it does not hold. Moreover, our proof system is not relatively complete if we omit this assumption. The example at the end of this section shows how the additional assumption is used in practice.

An Example with Dynamic Binding

In this section we describe a somewhat larger example proof outline and its resulting verification conditions. The example involves dynamic binding and heap modifications.

Consider the class *Clock* and its subclass *FastClock* in Figure 5.3. The subclass overrides the *tick* method: fast clocks run twice as fast as normal clocks. The methods have a specification that reflects their behavior. The annotation of the method bodies has been omitted.

Now assume that we want a method *doubleTick* that aims to increment the time of a clock by two. The method has to treat fast clocks differently because one call to their *tick* method suffices for fast clocks. The proof outline of this method is listed in Figure 5.4.

Note that the conditional statement in the method body has no else branch, which is illegal according to the cOOre syntax definition in Section 2.2.1. It would be a very simple task to add such a statement and define its proof obligations. We trust that it suffices to say here that the precondition of the conditional statement and the formula $!(c \ \mathsf{instanceof}\ \mathit{FastClock}) = \mathsf{false}$ must imply its postcondition.

```
class Clock {
  int time ;

  requires true;
  ensures this.time =
            old(this.time) + 1
  void tick() {
    this.time := this.time + 1 ;
  }
}
```

```
class FastClock extends Clock {

  requires true;
  ensures this.time =
            old(this.time) + 2
  void tick() {
    this.time := this.time + 2 ;
  }
}
```

Figure 5.3: Two Clocks

```
requires true;
ensures c.time = old(c.time) + 2;
void doubleTick(Clock c) {
  assert c.time = old(c.time);
  c.tick() ;
  assert c.time = old(c.time) + (c instanceof FastClock ? 2 : 1);
  if (!(c instanceof FastClock)) {
    c.tick() ;
  }
  assert c.time = old(c.time) + 2;
}
```

Figure 5.4: The proof outline of method *doubleTick*.

Each of the two method calls has two verification conditions, one for each implementation of the method. The first call $c.tick()$ has verification condition (5.25a) for the implementation in *Clock*, and (5.25b) for the implementation in *FastClock*.

$$\begin{aligned} & c \text{ instanceof } Clock \wedge \neg(c \text{ instanceof } FastClock) \wedge \mathsf{defined}(c) \wedge \neg(c = \mathsf{null}) \\ & \wedge \mathcal{H}(time_{Clock})[\mathsf{f}(c)] = \mathsf{old}(c.time) \\ & \wedge(\forall \mathsf{rec} : Clock \bullet \mathsf{rec} \in \mathcal{H} \rightarrow (\mathsf{true} \wedge \mathsf{rec} = c \rightarrow \mathsf{rec}.time = \mathcal{H}(time_{Clock})[\mathsf{f}(c)]+1)) \\ & \qquad \rightarrow c.time = \mathsf{old}(c.time) + (c \text{ instanceof } FastClock \ ? \ 2 : 1) \end{aligned} \quad (5.25a)$$

$$
\begin{aligned}
& c \text{ instanceof } \mathit{FastClock} \wedge \mathsf{defined}(c) \wedge \neg(c = \mathsf{null}) \\
& \qquad \wedge \mathcal{H}(\mathit{time}_{\mathit{Clock}})[\mathsf{f}(c)] = \mathsf{old}(c.\mathit{time}) \\
& \wedge (\forall \mathsf{rec} : \mathit{FastClock} \bullet \mathsf{rec} \in \mathcal{H} \rightarrow \\
& \quad (\mathsf{true} \wedge \mathsf{rec} = (\mathit{FastClock})c \rightarrow \mathsf{rec}.\mathit{time} = \mathcal{H}(\mathit{time}_{\mathit{Clock}})[\mathsf{f}(c)] + 2)) \\
& \qquad \rightarrow c.\mathit{time} = \mathsf{old}(c.\mathit{time}) + (c \text{ instanceof } \mathit{FastClock} \ ?\ 2 : 1) \qquad (5.25b)
\end{aligned}
$$

We omitted the heap_V predicates in both verifications conditions for brevity; it is the formula

$$\mathsf{heap} \wedge c \in \mathcal{H} \wedge \mathsf{this} \in \mathcal{H} \ .$$

for both verification conditions. With this additional assumption both verification conditions hold. The additional assumption $\neg(c \text{ instanceof } \mathit{FastClock})$ in (5.25a) turns out to be necessary in order to prove that the expression c instanceof $\mathit{FastClock}$? 2 : 1 has the value 1. Similarly, we need the assumption c instanceof $\mathit{FastClock}$ to reduce the same conditional expression to 2 in the second formula.

The verification conditions (5.26a) and (5.26b) of the second call are also interesting because they reveal a different way to use the information regarding the class of the receiver.

$$
\begin{aligned}
& c \text{ instanceof } \mathit{Clock} \wedge \neg(c \text{ instanceof } \mathit{FastClock}) \wedge \mathsf{defined}(c) \wedge \neg(c = \mathsf{null}) \\
& \wedge \mathcal{H}(\mathit{time}_{\mathit{Clock}})[\mathsf{f}(c)] = \mathsf{old}(c.\mathit{time}) + (c \text{ instanceof } \mathit{FastClock} \ ?\ 2 : 1) \\
& \wedge !c \text{ instanceof } \mathit{FastClock} \wedge (\forall \mathsf{rec} : \mathit{Clock} \bullet \\
& \quad \mathsf{rec} \in \mathcal{H} \rightarrow (\mathsf{true} \wedge \mathsf{rec} = c \rightarrow \mathsf{rec}.\mathit{time} = \mathcal{H}(\mathit{time}_{\mathit{Clock}})[\mathsf{f}(c)] + 1)) \\
& \qquad \rightarrow c.\mathit{time} = \mathsf{old}(c.\mathit{time}) + 2 \qquad (5.26a)
\end{aligned}
$$

$$
\begin{aligned}
& c \text{ instanceof } \mathit{FastClock} \wedge \mathsf{defined}(c) \wedge \neg(c = \mathsf{null}) \\
& \wedge \mathcal{H}(\mathit{time}_{\mathit{Clock}})[\mathsf{f}(c)] = \mathsf{old}(c.\mathit{time}) + (c \text{ instanceof } \mathit{FastClock} \ ?\ 2 : 1) \\
& \wedge !(c \text{ instanceof } \mathit{FastClock}) \wedge (\forall \mathsf{rec} : \mathit{FastClock} \bullet \mathsf{rec} \in \mathcal{H} \rightarrow \\
& \quad \mathsf{true} \wedge \mathsf{rec} = (\mathit{FastClock})c \rightarrow \mathsf{rec}.\mathit{time} = \mathcal{H}(\mathit{time}_{\mathit{Clock}})[\mathsf{f}(c)] + 2)) \\
& \qquad \rightarrow c.\mathit{time} = \mathsf{old}(c.\mathit{time}) + 2 \qquad (5.26b)
\end{aligned}
$$

Again, we have omitted the heap_V formula, which corresponds to the same formula as in the previous verification conditions. The antecedent of (5.26b) is a contradiction, which trivializes this verification condition. It corresponds to an implementation that will never be bound to this particular call due to the guard that protects it.

5.2.4 Soundness

We will prove the soundness of the adaptation rule for dynamically-bound method calls in this section. We start with a simple lemma that states that

the $\lfloor . \rfloor$ operator is type preserving.

Lemma 5.4. *For every assertion Q we have $[Q] = [\lfloor Q \rfloor]$.*

Proof. By structural induction on Q. □

There is a minor problem with the substitution $[e_0/\mathsf{this}]$ in (VC_j). Thus far, we have only used substitutions in which the variable that is being replaced has a type that is a supertype of the type of the expression that replaces it. However, $[e_0] \preceq [\mathsf{this}]$ need not hold for every verification condition in this rule; most of the classes C_j will be subtypes of $[e_0]$, and $[\mathsf{this}] = C_j$ in every method specification $\{P_j\}\ m@C_j\ \{Q_j\}$. The following variant of Lemma 4.3 covers this case.

Lemma 5.5. *For every assertion Q, every state (s, h), every compatible freeze state (s', h'), and every expression e such that $[e] \sim [\mathsf{this}]$, we have*

$$\mathcal{A}[\![Q[e/\mathsf{this}]]\!](s, h)(s', h') = \mathcal{A}[\![Q]\!](s[\mathsf{this} \mapsto v], h)(s', h')$$

if $\mathcal{E}[\![e]\!](s, h) = v \neq \bot$ and $v \in dom([\mathsf{this}])$.

Proof. By structural induction on Q. □

We also need a lemma that states that the heap increases monotonically during each computation.

Lemma 5.6. *For every computation $\langle S, (s, h)\rangle \to (s', h')$ of some statement S we have $dom(h) \subseteq dom(h')$.*

Proof. By induction on the length of the derivation. □

We now turn our attention to the relation between the dual heap and a computation. The following lemma essentially states that $\lfloor Q \rfloor$ holds in relation to the final heap of a computation if and only if Q holds in relation to the initial heap of the computation, provided that the variables of the heap model store the initial heap values.

Lemma 5.7. *Let Q be an arbitrary assertion. Let $\mathcal{H}$ be a fresh logical variable of type object^*. Let $\mathcal{H}(x_C)$ be a fresh logical variable of type t^*, for every instance variable $x : t$ declared in some class C which occurs in Q, and let f be a fresh function symbol. Let h, h' be two heaps such that $dom(h) \subseteq dom(h')$. Let s be a stack such that the following three formulas hold in the state (s, h).*

$$(\forall o : \mathsf{object} \bullet o = \mathsf{null} \vee o \in \mathcal{H}) \tag{5.27}$$

$$\mathsf{f}(\mathsf{null}) = \mathsf{length}(\mathcal{H}) \tag{5.28}$$

$$\bigwedge_{x_C} (\forall o : C \bullet \neg(o = \mathsf{null}) \to o.x = \mathcal{H}(x_C)[\mathsf{f}(o)]) \tag{5.29}$$

Finally, Let (s'', h'') be an arbitrary freeze state that is compatible with h. Then

$$(s, h)(s'', h'') \models Q \quad \Longleftrightarrow \quad (s, h')(s'', h'') \models \lfloor Q \rfloor \ .$$

Proof. By structural induction on Q. We first consider the most interesting case of an expression $q.x$ and prove $\mathcal{L}[\![q.x]\!](s^*,h)(s'',h'') = \mathcal{L}[\![\lfloor q.x \rfloor]\!](s^*,h')(s'',h'')$. Let $\mathsf{origin}([q],x) = C$. We have $\mathcal{L}[\![q]\!](s^*,h)(s'',h'') = \mathcal{L}[\![\lfloor q \rfloor]\!](s^*,h')(s'',h'')$ by the induction hypothesis. If $\mathcal{L}[\![q]\!](s^*,h)(s'',h'') \notin \{\bot, null\}$ then

$$
\begin{array}{ll}
 & \mathcal{L}[\![\lfloor q.x \rfloor]\!](s^*,h')(s'',h'') \\
= & \{\ \text{def. } \lfloor\rfloor\ \} \\
 & \mathcal{L}[\![\mathcal{H}(x_C)[\mathsf{f}(\lfloor q \rfloor)]]\!](s^*,h')(s'',h'') \\
= & \{\ \text{def. } \mathcal{L}[\![\_]\!],\ \text{where } s^*(\mathcal{H}(x_C)) = (g,n),\ \text{and } \mathcal{L}[\![\lfloor q \rfloor]\!](s^*,h')(s'',h'') \neq \bot\ \} \\
= & \{\ \text{def. } \mathcal{L}[\![\_]\!],\ \text{and ind. hyp.}\ \} \\
 & \mathcal{L}[\![\mathcal{H}(x_C)[\mathsf{f}(q)]]\!](s^*,h)(s'',h'') \\
= & \{\ (5.29)\ ,\ \mathcal{L}[\![q]\!](s^*,h)(s'',h'') \notin \{\bot, null\} \text{ and def. } \mathcal{L}[\![\_]\!]\ \} \\
 & \mathcal{L}[\![q.x]\!](s^*,h)(s'',h'')\ .
\end{array}
$$

If $\mathcal{L}[\![q]\!](s^*,h)(s'',h'') = \mathcal{L}[\![\lfloor q \rfloor]\!](s^*,h')(s'',h'') = null$ then

$$
\begin{array}{lll}
\mathcal{L}[\![\lfloor q.x \rfloor]\!](s^*,h')(s'',h'') & & \{\ \text{def. } \lfloor\rfloor\ \} \\
= \ \mathcal{L}[\![\mathcal{H}(x_C)[\mathsf{f}(\lfloor q \rfloor)]]\!](s^*,h')(s'',h'') & & \{\ \text{def. } \mathcal{L}[\![\_]\!] \text{ and } (5.28)\ \} \\
= \ \bot & & \{\ \text{def. } \mathcal{L}[\![\_]\!]\ \} \\
= \ \mathcal{L}[\![q.x]\!](s^*,h)(s'',h'')\ . & &
\end{array}
$$

Similarly, iff $\mathcal{L}[\![q]\!](s^*,h)(s'',h'') = \mathcal{L}[\![\lfloor q \rfloor]\!](s^*,h')(s'',h'') = \bot$ then

$$
\begin{array}{lll}
\mathcal{L}[\![\lfloor q.x \rfloor]\!](s^*,h')(s'',h'') & & \{\ \text{def. } \lfloor\rfloor\ \} \\
= \ \mathcal{L}[\![\mathcal{H}(x_C)[\mathsf{f}(\lfloor q \rfloor)]]\!](s^*,h')(s'',h'') & & \{\ \text{def. } \mathcal{L}[\![\_]\!]\ \} \\
= \ \bot & & \{\ \text{def. } \mathcal{L}[\![\_]\!]\ \} \\
= \ \mathcal{L}[\![q.x]\!](s^*,h)(s'',h'')\ . & &
\end{array}
$$

Another non-trivial case concerns assertions of the form $(\exists z : C \bullet Q)$. We must prove that

$$\mathcal{A}[\![(\exists z : C \bullet Q)]\!](s^*,h)(s'',h'') = \mathcal{A}[\![\lfloor(\exists z : C \bullet Q)\rfloor]\!](s^*,h')(s'',h'')\ .$$

We prove this case as follows.

$$
\begin{array}{ll}
 & \mathcal{A}[\![\lfloor(\exists z : C \bullet Q)\rfloor]\!](s^*, h')(s'', h'') \\
= & \{\ \text{def. } \lfloor\rfloor\ \} \\
 & \mathcal{A}[\![(\exists z \in \mathcal{H} : C \bullet \lfloor Q\rfloor)]\!](s^*, h')(s'', h'') \\
= & \{\ \text{def. bounded quantification}\ \} \\
 & \mathcal{A}[\![(\exists z : C \bullet (z = \mathsf{null} \vee z \in \mathcal{H}) \wedge \lfloor Q\rfloor)]\!](s^*, h')(s'', h'') \\
= & \{\ \text{def. } \mathcal{A}[\![\_]\!]\ \} \\
 & \alpha \in (\{null\} \cup rng(s^*(\mathcal{H}))) \text{ and} \\
 & \quad \mathcal{A}[\![\lfloor Q\rfloor]\!](s^*[z \mapsto \alpha], h')(s'', h'') = tt \text{ for some } \alpha \in dom(C) \cap dom(h') \\
= & \{\ (\{null\} \cup rng(s^*(\mathcal{H}))) \subseteq dom(h)\ \} \\
 & \alpha \in (\{null\} \cup rng(s^*(\mathcal{H}))) \text{ and} \\
 & \quad \mathcal{A}[\![\lfloor Q\rfloor]\!](s^*[z \mapsto \alpha], h')(s'', h'') = tt \text{ for some } \alpha \in dom(C) \cap dom(h) \\
= & \{\ \text{Eq. (5.27) and def. } \mathcal{A}[\![\_]\!]\ \} \\
 & \mathcal{A}[\![\lfloor Q\rfloor]\!](s^*[z \mapsto \alpha], h')(s'', h'') = tt \text{ for some } \alpha \in dom(C) \cap dom(h) \\
= & \{\ \text{ind. hyp.}\ \} \\
 & \mathcal{A}[\![Q]\!](s^*[z \mapsto \alpha], h)(s'', h'') = tt \text{ for some } \alpha \in dom(C) \cap dom(h) \\
= & \{\ \text{def. } \mathcal{A}[\![\_]\!]\ \} \\
 & \mathcal{A}[\![(\exists z : C \bullet Q)]\!](s^*, h)(s'', h'')
\end{array}
$$

□

Our next lemma describes the correspondence between the context switch and the substitution $[e_0, \bar{e}/\mathsf{this}, \bar{p}]$.

Lemma 5.8. *Let $\bar{p} = p_1, \ldots, p_n$ be a sequence of formal parameters, and let $\bar{e} = e_1, \ldots, e_n$ be a corresponding sequence of expressions with $[e_i] \preceq [p_i]$, for $i \in \{1 \ldots n\}$. Let e_0 be an expression such that $[\mathsf{this}] \sim [e_0]$. Then we have, for every precondition P, and every state (s, h) such that $\mathcal{E}[\![e_0]\!](s, h) \in dom([\mathsf{this}])$ and $\mathcal{E}[\![e_i]\!](s, h) = v_i \neq \bot$ for every $i \in \{0 \ldots n\}$, that*

$$(s[\mathsf{this}, \bar{p} \mapsto \bar{v}], h) \models P \quad \Longleftrightarrow \quad (s, h) \models P[e_0, \bar{e}/\mathsf{this}, \bar{p}]\ ,$$

where $\bar{v} = v_0, \ldots, v_n$.

Proof. By structural induction on P. The proof combines the proofs of Lemma 4.3 and Lemma 5.5. □

The following lemma combines Lemma 5.7 and Lemma 5.8 above in order to describe the effect of the operation $[\mathsf{g}(.)/\mathsf{old}(.)]$ on a formula.

Lemma 5.9. *Let Q be an arbitrary assertion. Let $\mathcal{H}$ and $\mathcal{H}(x_C)$ be as as in Lemma 5.7. Let $(s, h), \bar{p} = p_1, \ldots, p_n, \bar{e} = e_1, \ldots, e_n$, and e_0 be as in Lemma 5.8. Moreover, let $(s, h) \models (5.27)$, $(s, h) \models (5.28)$ and $(s, h) \models (5.29)$. Let h' be a heap such that $dom(h) \subseteq dom(h')$. Then*

$$(s, h')(s[\mathsf{this}, \bar{p} \mapsto \bar{v}], h) \models Q \text{ if and only if } (s, h') \models Q[\mathsf{g}(.)/\mathsf{old}(.)]\ ,$$

where g is the syntactical operation that takes a formula Q and returns the formula $\lfloor Q[e_0, \bar{e}/\mathsf{this}, \bar{p}]\rfloor$, and $\bar{v} = v_0, \ldots, v_n$.

Proof. By structural induction on Q. Observe that the value of $Q[\mathsf{g}(.)/\mathsf{old}(.)]$ does not depend on any freeze state because the operation $[\mathsf{g}(.)/\mathsf{old}(.)]$ removes all expressions of the form $\mathsf{old}(e)$. The only non-trivial case concerns a logical expression of the form $\mathsf{old}(e)$. We must prove that

$$\mathcal{L}[\![\mathsf{old}(e)]\!](s,h')(s[\mathsf{this},\bar{p}\mapsto\bar{v}],h) = \mathcal{N}[\![\mathsf{old}(e)[\mathsf{g}(.)/\mathsf{old}(.)]]\!](s,h') \ .$$

We compute this case as follows.

$\mathcal{N}[\![\mathsf{old}(e)[\mathsf{g}(.)/\mathsf{old}(.)]]\!](s,h')$	{ def. $[\mathsf{g}(.)/\mathsf{old}(.)]$ }
$\mathcal{N}[\![\mathsf{g}(e)]\!](s,h')$	{ def. g }
$\mathcal{N}[\![\lfloor e[e_0,\bar{e}/\mathsf{this},\bar{p}]\rfloor]\!](s,h')$	{ Lemma 5.7 }
$\mathcal{N}[\![e[e_0,\bar{e}/\mathsf{this},\bar{p}]]\!](s,h)$	{ Lemma 5.8 }
$\mathcal{N}[\![e]\!](s[\mathsf{this},\bar{p}\mapsto\bar{v}],h)$	{ Lemma 3.3 }
$\mathcal{E}[\![e]\!](s[\mathsf{this},\bar{p}\mapsto\bar{v}],h)$	{ def. $\mathcal{L}[\![\_]\!]$ }
$\mathcal{L}[\![\mathsf{old}(e)]\!](s,h')(s[\mathsf{this},\bar{p}\mapsto\bar{v}],h)$	

□

Our final lemma can be used to reason about the substitutions $[\mathsf{rec}/\mathsf{this}]$ and $[\mathsf{result}/u]$ in the verification conditions of the adaptation rule.

Lemma 5.10. *Let v be a local variable u or the receiver keyword* this. *Let z be a logical variable such that $[z] \preceq [v]$. Let Q be an assertion that has no subformulas of the form $(\exists z \bullet Q')$ or $(\forall z \bullet Q')$. Then we have, for every state (s,h) such that $s(z) = s(v)$, and every compatible freeze state (s',h'), that*

$$(s,h)(s',h') \models Q \text{ if and only if } (s,h)(s',h') \models Q[z/v] \ .$$

Proof. By structural induction on Q. □

We are now able to prove the main result of this section: the soundness of the adaptation rule.

Theorem 5.11. *The adaptation rule (5.24) is sound.*

Proof. Let $\mathsf{impls}([e_0],m) = \{C_1,\ldots,C_k\}$, and let $\models \{P_1\}\ m@C_1\ \{Q_1\},\ \ldots,$ $\models \{P_k\}\ m@C_k\ \{Q_k\}$. Moreover, let $\models VC_1, \ldots, \models VC_k$. We must prove that $\models \{Q\}\ u := e_0.m(e_1,\ldots,e_n)\ \{Q'\}$. Let $\langle u := e_0.m(e_1,\ldots,e_n),(s,h)\rangle \to (s',h')$, and let $(s,h)(s'',h'') \models Q$ for some freeze state (s'',h''). We must then prove that $(s',h')(s'',h'') \models Q'$.

The given computation of the call can only be derived using rule MC_1 of the operational semantics. Hence the following must hold.

$$\mathcal{E}[\![e_0]\!](s,h) = o = (C,i) \tag{5.30}$$

$$\mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \text{ for } i \in \{1\ldots n\} \tag{5.31}$$

$$\mathsf{meth}(C,m) \equiv t\ m(p_1,\ldots,p_n)\{\ S\ \mathsf{return}\ e\ \} \tag{5.32}$$

$$\langle S,(s[\mathsf{this},p_1,\ldots,p_n\mapsto o,v_1,\ldots,v_n],h)\rangle \to (s^\circ,h') \tag{5.33}$$

$$\mathcal{E}[\![e]\!](s^\circ,h') = v \neq \bot \tag{5.34}$$

$$s' = s[u\mapsto v] \tag{5.35}$$

Our next step is to build a stack in which the variables of the heap model have the same values as the corresponding locations in the initial heap h. Let $\bar{v}$ be an arbitrary sequence without repetitions that contains every object $o \in dom(h)$ (except *null*). Let $\bar{v}[i]$ denote the object at position i in $\bar{v}$. For every instance variable x declared in some class C, let $\bar{v}(x_C)$ be the sequence of the same length as $\bar{v}$ such that the i-th element of $\bar{v}(x_C)$ is $h(\bar{v}[i])(C)(x)$, if $\bar{v}[i] \in dom(C)$, and $\mathsf{init}([x])$ otherwise. Let $\hat{\mathsf{f}} : dom(\mathsf{object}) \to dom(\mathsf{int})$ denote a function such that $\hat{\mathsf{f}}(null)$ denotes the length of $\bar{v}$, and for every object stored at some index i of $\bar{v}$ we have $\hat{\mathsf{f}}(\bar{v}[i]) = i$. Let $s^\dagger$ be the stack that is obtained from s by assigning $\bar{v}$ to $\mathcal{H}$, $\hat{\mathsf{f}}$ to f, and $\bar{v}(x_C)$ to $\mathcal{H}(x_C)$, for every field x declared in some class C. Finally, let $s^* = s^\dagger[\mathsf{result} \mapsto \mathcal{E}[\![e]\!](s^\circ, h')]$.

Note that $C \preceq [e_0]$ (Lemma 2.4 and Lemma 2.2). Let $\mathsf{prov}(C, m) = C_j$. Then $C_j \in \{C_1, \ldots, C_k\}$, and $\mathsf{meth}(C, m)$ denotes the implementation of method m declared in class C_j. The verification condition that corresponds to the implementation in method C_j is

$$\begin{aligned} &\lfloor \mathsf{boundto}(e_0, C_j, m) \rfloor \wedge \lfloor (\bigwedge\nolimits_{i=0}^{n} \mathsf{defined}(e_i)) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q \rfloor \\ &\quad \wedge (\forall \bar{z} \in \mathcal{H} \bullet \mathsf{g}_j(P_j \wedge \mathsf{rec} = \mathsf{this}) \to (Q_j[\mathsf{rec}/\mathsf{this}][\mathsf{g}_j(.)/\mathsf{old}(.)])) \\ &\quad \to Q'[\mathsf{result}/u] \ . \end{aligned} \tag{5.36}$$

From $\models$ (5.36) follows that $(s^*, h')(s'', h'') \models$ (5.36). We will prove that the antecedent of (5.36) also holds in that state to obtain $(s^*, h')(s'', h'') \models Q'[\mathsf{result}/u]$.

In order to do so, we will first prove that for every formula Q^* in which the logical variables of the heap model and result do not occur we have that

$$(s, h)(s'', h'') \models Q^* \text{ implies } (s^*, h')(s'', h'') \models \lfloor Q^* \rfloor \ . \tag{5.37}$$

From $(s, h)(s'', h'') \models Q^*$ follows $(s^*, h)(s'', h'') \models Q^*$ by the construction of s^* whenever the logical variables of the heap model and result do not occur in Q^*. We also have $(s^*, h) \models$ (5.27), $(s^*, h) \models$ (5.28), and $(s^*, h) \models$ (5.29) by the construction of s^*. Note that (5.33) and Lemma 5.6 imply $dom(h) \subseteq dom(h')$. Hence, by Lemma 5.7, we get $(s^*, h')(s'', h'') \models \lfloor Q^* \rfloor$.

Let us now return to the antecedent of (5.36). Since $\mathsf{prov}(C, m) = C_j$ it must be that $o \in dom(C_j)$ (Lemma 2.2). Hence $(s, h)(s'', h'') \models e_0$ $\mathsf{instanceof}$ C_j. Moreover, for every class $D \in \mathsf{overrides}(C_j)(m)$ we have $C \not\preceq D$ because otherwise we would have $\mathsf{prov}(C, m) = D$ instead of $\mathsf{prov}(C, m) = C_j$. Hence we have $o \notin dom(D)$ (Lemma 2.2), and consequently $(s, h)(s'', h'') \models \neg e_0$ $\mathsf{instanceof}$ D. Therefore we have $(s, h)(s'', h'') \models \mathsf{boundto}(e_0, C_j, m)$, and by Eq. (5.37) then also $(s^*, h')(s'', h'') \models \lfloor \mathsf{boundto}(e_0, C_j, m) \rfloor$.

From (5.30) and (5.31) follows $(s, h)(s'', h'') \models \bigwedge_{i=0}^{n} \mathsf{defined}(e_i)$, and by (5.37) then $(s^*, h')(s'', h'') \models \lfloor \bigwedge_{i=0}^{n} \mathsf{defined}(e_i) \rfloor$. By (5.30) and (5.37) we also have $(s, h)(s'', h'') \models \lfloor \neg(e_0 = \mathsf{null}) \rfloor$. To prove that $(s^*, h')(s'', h'') \models \mathsf{heap}_V$ we consider its parts. Firstly, we have $(s^*, h')(s'', h'') \models \mathsf{heap}$ by the construction of s^* (heap only depends on the variables of the heap model). The variables in V reference objects in $dom(h)$ because s is consistent with h. By the construction of

s^* these values also occur in $\mathcal{H}$. We must also prove that $(s^*, h')(s'', h'') \models \lfloor Q \rfloor$. Note that we have $(s, h)(s'', h'') \models Q$. Hence $(s^*, h')(s'', h'') \models \lfloor Q \rfloor$ using (5.37).

We now turn our attention to the second line of (5.36). Let $\bar{z} = z_1, \ldots, z_m$. Let $\bar{\alpha} = \alpha_1, \ldots, \alpha_m$ be an arbitrary sequence of values such that $\alpha_i \in dom([z_i])$, for $i \in \{1 \ldots m\}$. Moreover, assume that

$$(s^*[\bar{z} \mapsto \bar{\alpha}], h')(s'', h'') \models \bar{z} \in \mathcal{H} \wedge \mathsf{g}(P_j \wedge \mathsf{rec} = \mathsf{this}) \ , \tag{5.38}$$

where g is the syntactical operation that takes a formula Q and returns the formula $\lfloor Q[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor$. In other words, we have

$$(s^*[\bar{z} \mapsto \bar{\alpha}], h')(s'', h'') \models \bar{z} \in \mathcal{H} \wedge \lfloor (P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}]) \rfloor \ . \tag{5.39}$$

At this point, we can again use Lemma 5.7 to obtain

$$(s^*[\bar{z} \mapsto \bar{\alpha}], h)(s'', h'') \models (P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}]$$

from (5.39) because the variables of the heap model do not occur in $\bar{z}$. Moreover, the variables of the heap model and result also do not occur in the formula $(P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}]$, and therefore

$$(s[\bar{z} \mapsto \bar{\alpha}], h)(s'', h'') \models (P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}] \ .$$

The validity of $(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}] \models (P_j \wedge \mathsf{rec} = \mathsf{this})$, where $\bar{v} = v_1, \ldots, v_n$, then follows from Lemma 5.8. The second clause of this formula means that $s[\bar{z} \mapsto \bar{\alpha}](\mathsf{rec}) = o$.

Next, observe that the computation in (5.33) entails that we also have

$$\langle S, (s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \rangle \to (s^\circ[\bar{z} \mapsto \bar{\alpha}], h')$$

because the existence of a computation does not depend on the values of logical variables. From our initial assumption $\models \{P_j\} m@C_j \{Q_j\}$ and Definition 3.3 then follows

$$(s^\circ[\bar{z} \mapsto \bar{\alpha}][\mathsf{result} \mapsto \mathcal{E}[\![e]\!](s^\circ[\bar{z} \mapsto \bar{\alpha}], h')], h')(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \models Q_j \ . \tag{5.40}$$

Next, observe that $s^\circ(\mathsf{this}) = o$ because assignments to this are not allowed, and that $s^\circ[\bar{z} \mapsto \bar{\alpha}](\mathsf{rec}) = o$ because $\mathsf{rec} \in \bar{z}$ and, as observed above, we have $s[\bar{z} \mapsto \bar{\alpha}](\mathsf{rec}) = o$. Then (5.40) and Lemma 5.10 imply

$$(s^\circ[\bar{z} \mapsto \bar{\alpha}][\mathsf{result} \mapsto \mathcal{E}[\![e]\!](s^\circ[\bar{z} \mapsto \bar{\alpha}], h')], h')(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \models Q_j[\mathsf{rec}/\mathsf{this}] \ . \tag{5.41}$$

The formula $Q_j[\mathsf{rec}/\mathsf{this}]$ contains no free occurrences of local variables, and every logical variable that occurs free in Q_j (except result) occurs in $\bar{z}$. Moreover, $s^*(\mathsf{result}) = \mathcal{E}[\![e]\!](s^\circ, h')$. Therefore

$$(s^*[\bar{z} \mapsto \bar{\alpha}], h')(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \models Q_j[\mathsf{rec}/\mathsf{this}]$$

must hold too. We can also replace the freeze stack s in the formula above by s^* because the only variables in Q_j whose values depend on the freeze state are this and $p_1, \ldots, p_n$. Subsequently, we can use Lemma 5.9 to obtain

$$(s^*[\bar{z} \mapsto \bar{\alpha}], h')(s'', h'') \models Q_j[\mathsf{rec}/\mathsf{this}][\mathsf{g}_j(.)/\mathsf{old}(.)] \ . \tag{5.42}$$

Equation (5.42) shows that the entire antecedent of (5.36) holds in the given state. Its consequent must then hold too: $(s^*, h')(s'', h'') \models Q'[\mathsf{result}/u]$. If we expand s^* we get $(s^\dagger[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'[\mathsf{result}/u]$. Then we know that

$$(s[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'[\mathsf{result}/u]$$

holds too because the dual heap variables do not occur in Q'. Moreover, $Q'[\mathsf{result}/u]$ does also not depend on the value of u. Hence we also have $(s'[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'[\mathsf{result}/u]$. Then $(s'[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'$ follows from the definition of s' and Lemma 5.10. This shows that our main goal $(s', h')(s'', h'') \models Q'$ also holds because result does not occur in Q'. □

The result that we have proved above differs in one important sense from the soundness results of the previous chapter: we have proved the soundness of a rule whose assumptions are not only verification conditions, but also entire method specifications of the form $\{P\}\ m@C\ \{Q\}$. The soundness of the complete proof outline logic therefore depend on an additional meta-proof that shows that the validity of the verification conditions of all methods also establishes the validity of the corresponding method annotations. We will present such an argument in Section 7.2.

5.3 Related Work

The rules for reasoning about method calls in this chapter are variants of the rules that we presented in two previous papers [PdB03b, PdB04]. We have shown elsewhere [PdB04] how the adaptation rule in this thesis can be optimized by taking the *effects* of a method into account. We will further discuss this topic in Section 8.4.2.

Poetzsch-Heffter and Müller [PHM99, Mül02] follow Gorelick's approach (see Section 5.1.1) in their logic, which does not lead to a proof outline logic (cf. Section 5.1.2). They have, for example, a separate substitution rule for reasoning about the local state of a caller.

Poetzsch-Heffter and Müller use syntactical constructs of the form $T : m$, which are called *virtual methods*, to structure reasoning about dynamically-bound method calls. A specification of a virtual method $T : m$ reflects the common properties of all implementations that might be bound to a call of method m on a receiver with static type T. Proving the specification of a virtual method proceeds in two steps. In the first step, one proves that the implementation of method m in class T satisfies the specification. In the second

step, one shows that the specification also holds for the virtual method $S : m$ of every subtype S of T.

Previous work on adaptation rules [Hoa71, Old83, BMW89, Nau00] focusses on imperative languages with global variables. The soundness of adaptation rules always depends on the state changes that method executions may cause, which implies that adaptation rules are necessarily tailored to a specific language.

Kleymann [Kle99] combines the adaptation rule with the rule of consequence. Thus he obtains a stronger rule of consequence that is also able to adapt the meaning of logical (or auxiliary) variables in specifications. Our proof outline logic does not have a rule of consequence, and we use our adaptation rule very economically. That is, we only use it to adapt method specifications in the context of method calls.

Homeier and Martin [HM03] also combine the adaptation rule with the call rule in their logic for basic recursive procedures. This enables them to define a verification condition generator (VCG).

Von Oheimb uses several variants of Kleymann's rule of consequence in his logic for Java [vO01]. His call rule quantifies universally over the dynamic type D of the receiver. The quantification range is, moreover, restricted to classes that are subtypes of the static type of the receiver. Our rule goes one step further by clustering the set of possible classes D that must be considered into groups that share a particular implementation.

Chapter 6

Reasoning about Object Creation

In the previous two chapters, we have analyzed how the data in object-oriented programs is manipulated by assignments, and how this process is controlled using method calls. We have seen how we can describe the behavior of methods using program annotation, and how we can compute the corresponding proof obligations that collectively ensure that the program annotation constitutes a valid proof outline.

In this chapter we turn our attention to the creation of objects, which is another key feature of object-oriented programming. Reasoning about object creation has no clear counterparts in the early work on Hoare logics which typically focussed on languages with a fix set of variables. By contrast, object-oriented programs may allocate new objects in each state, and every new object has its own set of instance variables.

Recall from Section 2.2.1 that a statement $u := \mathsf{new}\ C(e_1, \ldots, e_n)$ allocates a new object of class C, and subsequently calls the constructor method of class C on this new object; the expressions $e_1, \ldots, e_n$ denote the actual parameters of the call. Upon completion of the call, a reference to the new object is assigned to the local variable u. The allocation step creates the new object, whereas the constructor method typically initializes its fields with specific values.

The allocation of a new object and the subsequent call to the constructor method are two distinct steps, which are initiated by a single statement in our language cOOre. Combining these two steps in one statement ensures that each object is always immediately initialized after its allocation. However, the two steps differ greatly, and they require quite distinct reasoning techniques.

This chapter is therefore organized as follows. First, we explore reasoning about object allocation using a weakest precondition calculus (Section 6.1.1). Secondly, we show that we can also express the strongest postconditions of object allocations in our assertion language (Section 6.1.2). In Section 6.2, we

formulate an adaptation rule for reasoning about calls to constructor methods. Moreover, we also explain how this rule can be combined with the results of the previous section in order to reason about the total effect of a statement $u := \mathsf{new}\ C(e_1, \ldots, e_n)$. As usual, we discuss related work in the final section (6.3) of the chapter.

6.1 Object Allocation

Object creation starts with the allocation of a new object. Formally, object allocation means that a new object is added to the domain of the heap. The identity of the new object always differs from the identities of all previously allocated objects. The fields that are part of the internal state of the new object receive their default values during the allocation.

We model the allocation of a new object of some class C by means of a pseudo statement $u := \mathsf{alloc}(C)$. This statement allocates a new object of class C, and subsequently assigns the new object to the local variable u.

The effect of a statement $u := \mathsf{alloc}(C)$ can be described formally by means of an operational rule. The following rule models object allocation as a non-deterministic execution step because it does not fix the identity of the new object that is being allocated: every object that satisfies the premisses may be the object that is allocated. We prefer this abstract semantics over a more concrete semantics that enforces a particular allocation order.

$$\frac{o = (C, i) \qquad h(o) \text{ is undefined}}{\langle u := \mathsf{alloc}(C), (s, h)\rangle \rightarrow (s[u \mapsto o], h \cdot [o \mapsto \mathit{init}(C)])} \qquad (OA)$$

Recall from Section 2.2.3 that $\mathit{init}(C)$ denotes the initial internal state of an object of type C. We have $\mathit{init}(C)(D)(x) = \mathsf{init}([x])$ for every field $x \in \mathit{Fields}(D)$ declared in some class D such that $C \trianglelefteq D$; the function is undefined for all other fields.

6.1.1 A WP-Calculus for Object Allocation

In this section, we will define a weakest precondition calculus for statements of the form $u := \mathsf{alloc}(C)$. Unfortunately, we cannot simply use the substitution $[\mathsf{alloc}(C)/u]$ for reasoning about object allocation like we did in Chapter 4 for reasoning about basic assignments. For the expression $\mathsf{alloc}(C)$ is not a well-formed expression in our assertion language. It would, in principle, be possible to think of $\mathsf{alloc}(C)$ as denoting the new object *after* its allocation, but that does not answer the question what this expression denotes *prior* to the allocation. It cannot already denote the new object because this object does not exist prior to its allocation.

We will, nevertheless, define an operation $[\mathsf{new}(C)/u]$ in this section that computes the weakest precondition of the statement $u := \mathsf{alloc}(C)$ with respect to an arbitrary postcondition Q. This operation will be a *contextual* substitution

operation. Normal substitution operations simply replace the occurrences of a particular variable. A contextual substitution operation, however, substitutes the occurrences of a particular variable together with the immediate *contexts* in which they occur.

We will motivate the use of a contextual substitution operation by means of an example. Consider the postcondition $\neg(u = \mathsf{null})$. This postcondition holds after a statement $u := \mathsf{alloc}(C)$ because u references the new object when the statement terminates. Observe that we cannot replace the occurrence of u in the postcondition by the expression $\mathsf{alloc}(C)$ because the resulting formula $\neg(\mathsf{alloc}(C) = \mathsf{null})$ has no meaning in the initial state.

Recall that u is the only reference to the new object immediately after its allocation. Our contextual substitution operation $[\mathsf{new}(C)/u]$ will recognize the expression $u = \mathsf{null}$ as a possible context of the variable u. Secondly, it will recognize u as an expression which denotes the new object, and it will identify null as an expression that cannot denote the new object. Therefore it concludes that the value of the expression $u = \mathsf{null}$ must be *ff* in the postcondition, and it will substitute the entire expression $u = \mathsf{null}$ by the equivalent expression false. Thus the weakest precondition of the postcondition $\neg(u = \mathsf{null})$ becomes $\neg(\mathsf{false})$.

Atoms

We call an expression that is being replaced by our substitution operation an *atom*. An atom, like $u = \mathsf{null}$, always consists of an expression that denotes the new object (u in our example) together with its immediate context. However, there is another criterion for atoms, which is that only expressions that do *not* denote the new object itself are atoms. This is important because there is no expression that denotes this object in the initial state. And therefore we cannot replace such expressions by an expression that denotes the same value in the initial state.

A simple inspection of the typing rules of our assertion language reveals that expressions that denote the new object can only occur in the immediate context of a conditional expression, or an expression of one of the following forms.

$$(D)q \qquad q.x \qquad q \ \mathsf{instanceof}\ D \qquad q = q$$

Each of the subexpressions q in these expressions may denote the new object. Every expression of the last three forms is an atom of our contextual substitution operation if (one of its subexpressions) q indeed denotes the new object.

An expression $(D)q$ is only an atom if q denotes the new object, and if $C \not\preceq D$. Recall that C is the dynamic type of the new object. So the condition $C \not\preceq D$ predicts whether the cast in $(D)q$ will fail. If it fails, we know that $(D)q$ has the value $\bot$. Hence this condition ensures that $(D)q$ does not denote the new object itself. Conditional expressions are not atoms if their value is the new object. However, it turns out to be convenient to remove all conditional

expressions from an assertion before we compute its weakest precondition. We will explain how this can be done below.

Note that we can statically predict the value of each atom in the state immediately after the allocation of a new object. The value of an expression $u.x$, for example, is the default value of the type of field x because the fields of the new object initially have their default values. An expression u $\mathsf{instanceof}$ C is equivalent to true after executing $u := \mathsf{alloc}(C)$. We can also predict the value of an expression of the form $q = q'$ if either q or q' denotes the new object: its value is *tt* if both q and q' denote the new object, and *ff* if only one of these expressions denotes the new object.

All other operators in our language cannot be applied to the new object directly due to its type rules. Note also that an expression $\mathsf{old}(u)$ does *not* involve the new object because the occurrence of u in $\mathsf{old}(u)$ denotes an old value of u.

Removing Conditional Expressions

To precisely identify the set of atoms of our contextual substitution operation, we need to be able to determine statically which expressions denote the new object. A conditional expression like $\mathsf{true}\ ?\ u : u$ clearly also denotes the new object after the execution of $u := \mathsf{alloc}(C)$. However, we cannot statically determine whether an arbitrary conditional expression $q_1\ ?\ q_2 : q_3$ denotes the new object, because its value depends on the value of q_1.

We will therefore define a function that removes all conditional expressions from an assertion (except those inside expressions of the form $\mathsf{old}(e)$). This function is based on the following observation. Let Q be an arbitrary formula without quantifiers. Let $q \equiv q_1\ ?\ q_2 : q_3$ be a conditional expression in Q which does not occur inside an expression of the form $\mathsf{old}(e)$. Then Q is equivalent to

$$\begin{array}{ll} & q_1 = \mathsf{true} \rightarrow Q[(D)q_2/q] \\ \wedge & q_1 = \mathsf{false} \rightarrow Q[(D)q_3/q] \\ \wedge & \neg\mathsf{defined}(q_1) \rightarrow Q[(D)\mathsf{undefined}/q]\ , \end{array} \tag{6.1}$$

where $D = [q]$. By $[q'/q]$ we mean here the operation that replaces every occurrence of a logical expression q by another logical expression q'. To ensure that every formula $Q[q'/q]$ is well-typed we require that $[q] = [q']$.

The following lemma formally states the above-mentioned equivalence.

Lemma 6.1. *For every formula Q without quantifiers, every state (s, h), and every compatible freeze state (s', h'), we have*

$$(s, h)(s', h') \models Q \text{ if and only if } (s, h)(s', h') \models (6.1)\ .$$

Proof. By structural induction on Q. For the base case we must prove that for every expression q', every conditional expression $q \equiv q_1\ ?\ q_2 : q_3$, every state (s, h), and every freeze state (s', h'), we have

1. $\mathcal{L}[\![q_1]\!](s,h)(s',h') = tt \Longrightarrow v = \mathcal{L}[\![q'[(D)q_2/q]]\!](s,h)(s',h')$
2. $\mathcal{L}[\![q_1]\!](s,h)(s',h') = ff \Longrightarrow v = \mathcal{L}[\![q'[(D)q_3/q]]\!](s,h)(s',h')$
3. $\mathcal{L}[\![q_1]\!](s,h)(s',h') = \bot \Longrightarrow v = \mathcal{L}[\![q'[(D)\mathsf{undefined}/q]]\!](s,h)(s',h')$

where $v = \mathcal{L}[\![q']\!](s,h)(s',h')$ and $D = [q]$. All three claims can be proved easily by structural induction on q'. □

We will define a function $\mathsf{h}^* : \mathrm{QForm} \to \mathrm{QForm}$ that repeatedly performs this rewriting step until all conditional expressions have been removed from a formula. But first we define a function h that executes this step once. Let Q_0 be an arbitrary formula. If Q_0 contain no conditional expressions then $\mathsf{h}(Q_0) = Q_0$. Else, let $q \equiv q_1 \ ?\ q_2 : q_3$ be the leftmost innermost occurrence of a conditional expression in Q_0 (outside expressions of the form $\mathsf{old}(e)$). Moreover, let Q be the smallest subformula of Q_0 such that q occurs in Q'. Then $\mathsf{h}(Q_0)$ is the formula that is obtained by replacing Q in Q_0 by (6.1), where $D = [q]$. Note that Q is always a formula without quantifiers because a formula Q' is always smaller than a formula $(\exists z \bullet Q')$.

We have a lemma that states that h preserves the meaning of a formula.

Lemma 6.2. *For every assertion Q, every state (s,h), and every compatible freeze state (s',h') we have*

$$(s,h)(s',h') \models Q \text{ if and only if } (s,h)(s',h') \models \mathsf{h}(Q) \ .$$

Proof. By structural induction on Q. For the base case we need Lemma 6.1. □

Next, we define a family of functions h_n for $n \in \mathbb{N}$ such that h_0 is the identity function, and $\mathsf{h}_{n+1}(Q) = \mathsf{h}(\mathsf{h}_n(Q))$. Finally, we define h^* as the function such that, for every formula Q, we have $\mathsf{h}^*(Q) = \mathsf{h}_n(Q)$, where n is the smallest natural number such that $\mathsf{h}_n(Q)$ is a fixed point of h.

Note that every formula without conditional expressions is a fixed point of h. The function h^* is therefore well-defined if we can show that for each formula Q there exists an n such that $\mathsf{h}_n(Q)$ is a formula without conditional expressions. This is not immediately clear because an application of h to a formula may increase the total number of conditional expressions in a formula. For example, formula (6.1) contains more conditional expressions than Q itself if Q contains more than one conditional expression.

We can solve this issue by assigning a weight w to every formula. We define $\mathsf{w}(Q)$ by induction on the structure of Q. We have $\mathsf{w}(q) = (n+4)!$, where n is the number of conditional expressions in q. The other cases are straightforward.

$$\begin{array}{rcl} \mathsf{w}(\neg(Q)) & = & \mathsf{w}(Q) \\ \mathsf{w}(Q_1 \wedge Q_2) & = & \mathsf{w}(Q_1) + \mathsf{w}(Q_2) \\ \mathsf{w}((\exists z \bullet Q)) & = & \mathsf{w}(Q) \end{array}$$

Then we can prove that the application of h to a formula always decreases the weight of a formula. Let n be the number of conditional expressions in the

formula Q in the context of (6.1). The weight of (6.1) is then $3 \times ((n-1)+4)!$. It is not difficult to see that $n! > 3 \times (n-1)!$ for every $n \geq 4$. Hence

$$3 \times ((n-1)+4) = 3 \times ((n+4)-1)! < (n+4)! \ .$$

Lemma 6.3. *For every assertion Q we have that $\mathsf{h}^*(Q)$ is a formula without conditional expressions.*

Proof. Along the lines of the given proof sketch. □

Naturally, we also want h^* to preserve the meaning of a formula.

Lemma 6.4. *For every assertion Q, every state (s,h), and every compatible freeze state (s',h') we have*

$$(s,h)(s',h') \models Q \text{ if and only if } (s,h)(s',h') \models \mathsf{h}^*(Q) \ .$$

Proof. A simple consequence of Lemma 6.2. □

It is not strictly necessary to remove all conditional expressions from a formula in order to compute its weakest precondition. Elsewhere [PdB03b, PdB03c], we give an alternative definition of a weakest precondition calculus for object allocation which correctly handles conditional expressions. However, the calculus that we present in the remainder of this section is much easier to understand.

A clear disadvantage of the function h is that it increases the length of formulas. This makes it harder for human beings to read the resulting formulas. However, the case analysis which it performs is probably similar to the way in which a theorem prover would try to prove the validity of formulas with conditional expressions. This suggests that h^* merely implements a proof step that would otherwise be executed by the theorem prover!

Nuclei

We now return to our discussion of the atoms of our contextual substitution operation. We have observed that in order to correctly identify all atoms we have to be able to check statically whether a particular expression denotes the new object. We will call an expression that denotes the new object a *nucleus*.

Recall that u is the only program variable that references the new object after its allocation by the statement $u := \mathsf{alloc}(C)$. However, we have also seen that expressions like $(C)u$ denote the same value. We will prove that all other expressions cannot denote the new object (under some assumptions which we clarify below). Most expressions cannot denote the new object because the domain of their type does not include the new object. We will discuss the remaining cases in more detail.

A logical variable z can only reference the new object if it occurs inside a formula $(\exists z \bullet Q)$. For free occurrences of logical variables never refer to the new

object because the new object is only assigned to u. We will revisit this issue when we define the result of our contextual substitution operation on formulas of the form $(\exists z \bullet Q)$. We will then take measures which ensure that it is safe to assume that z never denotes the new object. A similar remark can be made about expressions of the form $z[q]$. We also assume that all formulas do not contain conditional expressions.

If these assumptions hold, then we can define the set of nuclei using the following grammar.

$$\nu \in \mathit{Nuclei} ::= u \mid (D)\nu$$

We assume here that $C \preceq D$. Note that the value of an expression $(D)q$ can only be the new object if $C \preceq D$ because otherwise the cast will fail whenever q references the new object.

The lemma below justifies our definition of the set of nuclei.

Lemma 6.5. *Let q be a logical expression in which no conditional expressions occur outside expressions of the form $\mathsf{old}(e)$. Furthermore, let (s, h) be an arbitrary state, and let (s', h') be a compatible freeze state. Let $o = (C, i)$ be such that $h(o)$ is undefined. Then*

$$\mathcal{L}[\![q]\!](s'', h'')(s', h') = o \text{ if and only if } q \in \mathit{Nuclei} \ ,$$

where $s'' = s[u \mapsto o]$ and $h'' = h \cdot [o \mapsto \mathit{init}(C)]$.

Proof. By structural induction on q. Note that u is the only reference to o in the new state because o does not exist in the state (s, h). □

Contextual Substitution in Expressions

We will define the result of $q[\mathsf{new}(C)/u]$ by induction on the structure of a logical expression q. Because $[\mathsf{new}(C)/u]$ is a contextual substitution which substitutes atoms (of which the nuclei are always a part) we do not have to define the result of $\nu[\mathsf{new}(C)/u]$. That is, $q[\mathsf{new}(C)/u]$ is not defined for expressions q which denote the new object. However, we will define the result of $q[\mathsf{new}(C)/u]$ for all atoms, and for all other logical expressions.

We will use in our definition of $q[\mathsf{new}(C)/u]$ the convention that q' and q'' denote arbitrary logical expressions that are *not* nuclei. The simple cases of $q[\mathsf{new}(C)/u]$ which do not involve an atom are gathered in Figure 6.1. We will discuss the other cases in more detail.

The first interesting case is $q.x[\mathsf{new}(C)/u]$ because q can be a nucleus. We have

$$\nu.x[\mathsf{new}(C)/u] \equiv \mathsf{init}([\nu.x])$$

because we know that every field of the new object has its default value after the allocation, and $\mathsf{init}([\nu.x])$ denotes this value. This case nicely illustrates our approach: our contextual substitution replaces atoms by expressions that have the same values. To ensure that the operation is type-preserving we have $\nu.x[\mathsf{new}(C)/u] \equiv (D)\mathsf{init}(D)$, where $D = [\nu.x]$, if D is a reference type. If q is

$$
\begin{array}{rcl}
\mathsf{null}[\mathsf{new}(C)/u] & \equiv & \mathsf{null} \\
\mathsf{this}[\mathsf{new}(C)/u] & \equiv & \mathsf{this} \\
u'[\mathsf{new}(C)/u] & \equiv & u' \quad (u' \not\equiv u) \\
\mathsf{op}(q_1,\ldots,q_n)[\mathsf{new}(C)/u] & \equiv & \mathsf{op}(q_1[\mathsf{new}(C)/u],\ldots,q_n[\mathsf{new}(C)/u]) \\
\mathsf{undefined}[\mathsf{new}(C)/u] & \equiv & \mathsf{undefined} \\
\mathsf{defined}(q)[\mathsf{new}(C)/u] & \equiv & \mathsf{defined}(q[\mathsf{new}(C)/u]) \\
z[\mathsf{new}(C)/u] & \equiv & z \quad (z \not\equiv u) \\
\mathsf{old}(e)[\mathsf{new}(C)/u] & \equiv & \mathsf{old}(e) \\
z[q][\mathsf{new}(C)/u] & \equiv & z[q[\mathsf{new}(C)/u]] \\
\mathsf{length}(z)[\mathsf{new}(C)/u] & \equiv & \mathsf{length}(z)
\end{array}
$$

Figure 6.1: Selected simple cases of the weakest precondition operation of object allocation.

not a nucleus we have $q'.x[\mathsf{new}(C)/u] \equiv (q'[\mathsf{new}(C)/u]).x$. The fact that q' is not a nucleus ensures that $q'[\mathsf{new}(C)/u]$ is defined.

The definition of q $\mathsf{instanceof}$ $D[\mathsf{new}(C)/u]$ requires a similar case split. If q is a nucleus, we can again predict the value of the atom in the final state. The second equation describes all cases where q is not a nucleus.

$$
\begin{array}{rcl}
\nu\ \mathsf{instanceof}\ D[\mathsf{new}(C)/u] & \equiv & \begin{cases} \mathsf{true} & \text{if } C \preceq D \\ \mathsf{false} & \text{otherwise} \end{cases} \\
q'\ \mathsf{instanceof}\ D[\mathsf{new}(C)/u] & \equiv & (q'[\mathsf{new}(C)/u])\ \mathsf{instanceof}\ D
\end{array}
$$

The definition of $(q = q)[\mathsf{new}(C)/u]$ is interesting because it often yields a result that is simpler than the original expression. We can distinguish four cases.

$$
\begin{array}{rcl}
(\nu = \nu)[\mathsf{new}(C)/u] & \equiv & \mathsf{true} \\
(\nu = q')[\mathsf{new}(C)/u] & \equiv & \mathsf{false} \\
(q' = \nu)[\mathsf{new}(C)/u] & \equiv & \mathsf{false} \\
(q' = q'')[\mathsf{new}(C)/u] & \equiv & (q'[\mathsf{new}(C)/u]) = (q''[\mathsf{new}(C)/u])
\end{array}
$$

Our final case concerns expressions of the form $(D)q$. We have

$$
\begin{array}{rcl}
(D)\nu[\mathsf{new}(C)/u] & \equiv & (D)\mathsf{undefined} \quad \text{if } C \not\preceq D \\
(D)q'[\mathsf{new}(C)/u] & \equiv & (D)(q'[\mathsf{new}(C)/u])
\end{array}
$$

Note that we do not have to define $(D)\nu[\mathsf{new}(C)/u]$ if $C \preceq D$ because in that case $(D)\nu \in \mathit{Nuclei}$.

The following lemma states that $q[\mathsf{new}(C)/u]$ is defined for every logical expression q that is not a nucleus. It implies that $q[\mathsf{new}(C)/u]$ is defined for every expression of type $\mathsf{boolean}$, which is important because only expressions of type $\mathsf{boolean}$ can be used in our assertion language as postconditions.

Lemma 6.6. *For every logical expression q in which no conditional expressions occur outside expressions of the form $\mathsf{old}(e)$ we have that $q[\mathsf{new}(C)/u]$ is defined if and only if $q \notin \mathit{Nuclei}$.*

Proof. By structural induction on q. □

The operator is also type-preserving.

Lemma 6.7. *Let q be a logical expression such that $q[\mathsf{new}(C)/u]$ is defined. Then $[q[\mathsf{new}(C)/u]] = [q]$.*

Proof. By structural induction on q. □

Finally, we have a lemma that states that $q[\mathsf{new}(C)/u]$ denotes the same value in the initial state as q in the state after execution of $u := \mathsf{alloc}(C)$.

Lemma 6.8. *Let q be a logical expression such that $q[\mathsf{new}(u)/C]$ is defined. Let (s, h) be an arbitrary state, and let (s', h') be a compatible freeze state. Let $o = (C, i)$ be such that $h(o)$ is undefined. Then*

$$\mathcal{L}[\![q[\mathsf{new}(C)/u]]\!](s,h)(s',h') = \mathcal{L}[\![q]\!](s'',h'')(s',h') \ ,$$

where $s'' = s[u \mapsto o]$ and $h'' = h \cdot [o \mapsto \mathit{init}(C)]$.

Proof. By induction on the complexity of q. We will discuss two illustrative cases here. For $q \equiv z$ we have $z[\mathsf{new}(C)/u] \equiv z$. By the construction of s'' we have $s(z) = s''(z)$. So we can reason as follows.

$$\begin{aligned}\mathcal{L}[\![z[\mathsf{new}(C)/u]]\!](s,h)(s',h') = \mathcal{L}[\![z]\!](s,h)(s',h') = s(z) = s''(z) \\ = \mathcal{L}[\![z]\!](s'',h'')(s',h')\end{aligned}$$

Next, we will consider all four cases of $q \equiv q_1 = q_2$. First, let $q \equiv \nu_1 = \nu_2$. Lemma 6.5 implies that $\mathcal{L}[\![\nu_1]\!](s'',h'')(s',h') = \mathcal{L}[\![\nu_2]\!](s'',h'')(s',h') = o$. Then

$$\begin{aligned}\mathcal{L}[\![(\nu_1 = \nu_2)[\mathsf{new}(C)/u]]\!](s,h)(s',h') = \mathcal{L}[\![\mathsf{true}]\!](s,h)(s',h') = tt \\ = \mathcal{L}[\![\nu_1 = \nu_2]\!](s'',h'')(s',h') \ .\end{aligned}$$

Secondly, let $q \equiv \nu = q'$. By definition $q' \notin \mathit{Nuclei}$. Hence Lemma 6.5 implies $\mathcal{L}[\![q']\!](s'',h'')(s',h') \neq o$. By the same lemma we get $\mathcal{L}[\![\nu]\!](s'',h'')(s',h') = o$. So

$$\begin{aligned}\mathcal{L}[\![(\nu = q')[\mathsf{new}(C)/u]]\!](s,h)(s',h') = \mathcal{L}[\![\mathsf{false}]\!](s,h)(s',h') = ff \\ = \mathcal{L}[\![\nu = q']\!](s'',h'')(s',h') \ .\end{aligned}$$

The case $q \equiv q' = \nu$ is similar. Finally, let $q \equiv q' = q''$. This case follows immediately from the induction hypothesis. Note that Lemma 6.6 ensures that $q'[\mathsf{new}(C)/u]$ and $q''[\mathsf{new}(C)/u]$ are defined. □

The Weakest Preconditions of Formulas

The extension of $[\mathsf{new}(C)/u]$ to formulas is straightforward in most cases. Observe that a formula can never be a nucleus because a formula has type boolean.

$$\begin{array}{rcl}
(\neg Q)[\mathsf{new}(C)/u] & \equiv & \neg(Q[\mathsf{new}(C)/u]) \\
(Q \wedge Q')[\mathsf{new}(C)/u] & \equiv & (Q[\mathsf{new}(C)/u]) \wedge (Q'[\mathsf{new}(C)/u]) \\
(\exists z : t \bullet Q)[\mathsf{new}(C)/u] & \equiv & (\exists z : t \bullet Q[\mathsf{new}(C)/u]), \text{ if } t \in \{\mathsf{int}, \mathsf{boolean}\} \\
(\exists z : t^* \bullet Q)[\mathsf{new}(C)/u] & \equiv & (\exists z : t^* \bullet Q[\mathsf{new}(C)/u]), \text{ if } t \in \{\mathsf{int}, \mathsf{boolean}\}
\end{array}$$

However, quantification over objects requires some care because object allocation may extend the domain of quantification. The domain of quantification of a formula $(\exists z : D \bullet Q)$ is extended if the formula quantifies over a supertype D of the class C of the new object. Consider, for example, the formula $(\exists z : C \bullet \neg(z = \mathsf{null}))$. The new object ensures that this formula holds in the extended state.

A formula $(\exists z : D \bullet Q)$ may hold in the postcondition for either the new object or some old object. For this reason, the weakest precondition of $(\exists z : D \bullet Q)$ has two clauses, which correspond to these two options.

$$(\exists z : D \bullet Q)[\mathsf{new}(C)/u] \equiv \left\{ \begin{array}{ll} (\exists z : D \bullet Q[\mathsf{new}(C)/u]) \vee (Q[u/z][\mathsf{new}(C)/u]) & \text{if } C \preceq D \\ (\exists z : D \bullet Q[\mathsf{new}(C)/u]) & \text{otherwise} \end{array} \right.$$

The first case corresponds to a domain extension. The first disjunct of this case represents the possibility that P holds for an old object, whereas the second disjunct covers the possibility that P holds for the new object. Recall that our contextual substitution operation assumes that occurrences of the logical variable z do not reference the new object. This assumption gives the first disjunct its intended meaning. In the second disjunct we want z to denote the new object. This is achieved by replacing z by u in that clause.

Example 6.1. *Consider the formula* $(\exists z : C \bullet \neg(z = \mathsf{null}))$*, which states that there exists an object of class* C *(or some subtype of class* C*). This formula clearly holds after the allocation of an object of class* C*. Therefore we expect that its weakest precondition with respect to the statement* $u := \mathsf{alloc}(C)$ *is equivalent to* true*. We can compute this weakest precondition as follows.*

$$\begin{array}{l}
(\exists z : C \bullet \neg(z = \mathsf{null}))[\mathsf{new}(C)/u] \\
\equiv (\exists z : C \bullet \neg(z = \mathsf{null})[\mathsf{new}(C)/u]) \vee \neg(z = \mathsf{null})[u/z][\mathsf{new}(C)/u] \\
\equiv (\exists z : C \bullet \neg((z = \mathsf{null})[\mathsf{new}(C)/u])) \vee \neg(u = \mathsf{null})[\mathsf{new}(C)/u] \\
\equiv (\exists z : C \bullet \neg((z[\mathsf{new}(C)/u]) = (\mathsf{null}[\mathsf{new}(C)/u]))) \vee \neg((u = \mathsf{null})[\mathsf{new}(C)/u]) \\
\equiv (\exists z : C \bullet \neg(z = \mathsf{null})) \vee \neg(\mathsf{false})
\end{array}$$

The resulting formula is indeed equivalent to true*.*

If a formula of the form $(\exists z : D^* \bullet Q)$ is valid in the state after an object allocation, and if $C \preceq D$, then it holds for a sequence of objects that may include

the new object at various positions. This means that there may be expressions of the form $z[i]$ in Q that denote the new object, which would undermine our definition of the set of nuclei. We solve this issue as follows. Note that we can construct a boolean sequence z' of the same length as z such that $z'[i] = \mathsf{true}$ holds if and only if the object at position i in the sequence of objects happens to be the new object. For this sequence, we have that

$$z[i] = (z'[i] ? u : z[i]) \ .$$

The expression on the right-hand side of this equation has the advantage that its subexpression $z[i]$ is not evaluated if it denotes the new object. For the expression u determines the value of the conditional expression if $z'[i]$ holds. Thus this equivalence reveals a way to reestablish our assumption in every expression on which the value of the weakest precondition depends.

We use an additional postfix operation $\langle z \|^u_{z'} \rangle$ to replace expressions of the form $z[i]$ by $(z'[i] ? u : z[i])$. Its two characteristic cases are listed below.

$$\begin{array}{rcl} \mathsf{length}(z)\langle z\|^u_{z'}\rangle & \equiv & \mathsf{length}(z) \\ (z[q])\langle z\|^u_{z'}\rangle & \equiv & z'[q\langle z\|^u_{z'}\rangle] \ ? \ (D)u : z[q\langle z\|^u_{z'}\rangle], \text{ where } D = [z[q]] \end{array}$$

The cast in $(D)u$ can be omitted if $[u] = D$. The operation $\langle z\|^u_{z'}\rangle$ is defined for the remaining expressions and extended to assertions in the standard way. It is not defined on z, but z can only occur in the two contexts that are described above.

We would like to point out here that our choice to support only two operators on sequences (indexing and the $\mathsf{length}(.)$ operator) ensures that the definition of $q\langle z\|^u_{z'}\rangle$ is straightforward. Allowing other operators on sequences (like concatenation) would complicate the definition of this operation. However, most operators on sequences can be encoded in terms of our two basic operators.

With the operation $\langle z \, \|^u_{z'} \rangle$ we can define the final case of $Q[\mathsf{new}(C)/u]$. The idea behind the definition of $(\exists z : D^* \bullet Q)[\mathsf{new}(C)/u]$ is that the formula $(\exists z : D^* \bullet Q)[\mathsf{new}(C)/u]$ holds in the final state if and only if there exists a sequence z' as described above which can be used to do the replacement.

$$(\exists z : D^* \bullet Q)[\mathsf{new}(C)/u] \equiv \begin{cases} (\exists z \bullet (\exists z' \bullet \mathsf{length}(z) = \mathsf{length}(z') \\ \qquad \wedge \ (\mathsf{h}^*(Q\langle z\|^u_{z'}\rangle)[\mathsf{new}(C)/u]))) & \text{if } C \preceq D \\ (\exists z : D^* \bullet Q[\mathsf{new}(C)/u]) & \text{otherwise} \end{cases}$$

We have to insert the function h^* in the right-hand side of the equation because $\langle z\|^u_{z'}\rangle$ may insert new conditional expressions in Q.

The following lemma shows that $Q\langle z\|^u_{z'}\rangle$ denotes the same value as Q provided that the values of z, z' and u are as described above.

Lemma 6.9. *Let Q be an arbitrary assertion in which the logical variable z' : $\mathsf{boolean}^*$ does not occur. Let u be a local variable such that $C \preceq [u]$. Let $z : D^*$ be a logical variable such that $C \preceq D$. Let (s, h) be a state, and let (s', h') be a compatible freeze state. Let $o = (C, i)$ be such that $h(o)$ is undefined. Let $\bar{v}$ be*

a sequence such that $\bar{v} \in dom(D^*)$ *and* $rng(\bar{v}) \subseteq dom(h)$. *Let* $\bar{v}'$ *be a sequence that is obtained from* $\bar{v}$ *by replacing an arbitrary number of elements in* $\bar{v}$ *by* o. *Let* $\bar{\beta}$ *be a boolean sequence of the same length as* $\bar{v}$ *such that the* i*-th element of* $\bar{\beta}$ *is* tt *if and only if the* i*-th element of* $\bar{v}'$ *is* o. *Then*

$$\mathcal{A}[\![Q\langle z\|^u_{z'}\rangle]\!](s''[z, z' \mapsto \bar{v}, \bar{\beta}], h'')(s', h') = \mathcal{A}[\![Q]\!](s''[z \mapsto \bar{v}'], h'')(s', h') \ ,$$

where $s'' = s[u \mapsto o]$ *and* $h'' = h \cdot [o \mapsto init(C)]$.

Proof. By structural induction on Q. For the base case we must prove that, for every logical assertion $q \not\equiv z$,

$$\mathcal{L}[\![q\langle z\|^u_{z'}\rangle]\!](s''[z, z' \mapsto \bar{v}, \bar{\beta}], h'')(s', h') = \mathcal{L}[\![q]\!](s''[z \mapsto \bar{v}'], h'')(s', h') \ ,$$

by structural induction on q. We will tackle the case $q \equiv z[q']$, which is the only interesting case of both claims. Recall that $z[q']\langle z\|^u_{z'}\rangle$ is syntactically equivalent to $z'[q'\langle z\|^u_{z'}\rangle]$? $(D)u : z[q'\langle z\|^u_{z'}\rangle]$. So we must prove that

$$\begin{aligned}\mathcal{L}[\![z'[q'\langle z\|^u_{z'}\rangle] \ ? \ (D)u : z[q'\langle z\|^u_{z'}\rangle]]\!](s''[z, z' \mapsto \bar{v}, \bar{\beta}], h'')(s', h') \\ = \mathcal{L}[\![z[q']]\!](s''[z \mapsto \bar{v}'], h'')(s', h') \ . \quad (6.2)\end{aligned}$$

By the induction hypothesis we have

$$\mathcal{L}[\![q'\langle z\|^u_{z'}\rangle]\!](s''[z, z' \mapsto \bar{v}, \bar{\beta}], h'')(s', h') = \mathcal{L}[\![q']\!](s''[z \mapsto \bar{v}'], h'')(s', h') \ .$$

Let $i = \mathcal{L}[\![q']\!](s''[z \mapsto \bar{v}'], h'')(s', h')$. If i is out of bounds for $\bar{v}$, then i is also out of bounds for $\bar{v}'$ and $\bar{\beta}$. Hence both sides of the equation in (6.2) evaluate to $\perp$. For all other indices i we must distinguish two cases. First, let $\mathcal{L}[\![z[q']]\!](s''[z \mapsto \bar{v}'], h'')(s', h') = o$. From $s''(u) = o$ and $C \preceq D$ follows that $\mathcal{L}[\![(D)u]\!](s''[z, z' \mapsto \bar{v}, \bar{\beta}], h'')(s', h') = o$. Moreover, for this case we have

$$\mathcal{L}[\![z'[q'\langle z\|^u_{z'}\rangle]]\!](s''[z, z' \mapsto \bar{v}, \bar{\beta}], h'')(s', h') = tt$$

by the construction of $\bar{\beta}$ and the induction hypothesis. Then clearly (6.2). In the second (and final) case we have $\mathcal{L}[\![z[q']]\!](s''[z \mapsto \bar{v}'], h'')(s', h') \neq o$. This means that the object at position i in the sequence $\bar{v}'$ is not o. Therefore it is equal to the object at position i in $\bar{v}$. Moreover, for this case we have

$$\mathcal{L}[\![z'[q'\langle z\|^u_{z'}\rangle]]\!](s''[z, z' \mapsto \bar{v}, \bar{\beta}], h'')(s', h') = ff$$

by the construction of $\bar{\beta}$ and the induction hypothesis. Equation (6.2) then follows from the definition of $\mathcal{L}[\![\_]\!]$. □

The most important result of this section is a lemma that states that, for every assertion Q without conditional expressions outside expressions of the form $\mathsf{old}(e)$, $Q[\mathsf{new}(C)/u]$ denotes the same value before $u := \mathsf{alloc}(C)$ as Q after the allocation. It will enable us to prove that $\mathsf{h}^*(Q)[\mathsf{new}(C)/u]$ is the weakest precondition of $u := \mathsf{alloc}(C)$ with respect to Q.

Lemma 6.10. *Let Q be an arbitrary assertion without conditional expressions outside expressions of the form* $\mathsf{old}(e)$*. Let (s,h) be an arbitrary state, and let (s',h') be a compatible freeze state. Let $o=(C,i)$ be such that $h(o)$ is undefined. Then*

$$\mathcal{A}[\![Q[\mathsf{new}(C)/u]]\!](s,h)(s',h') = \mathcal{A}[\![Q]\!](s[u\mapsto o], h\cdot[o\mapsto \mathit{init}(C)])(s',h') \ .$$

Proof. By induction on the complexity of Q. A non-standard complexity measure is required to prove the theorem because the operations h^* and $\langle z\|^u_{z'}\rangle$ may increase the length of a formula.[1]

We will consider the most interesting case, which concerns an assertion of the form $(\exists z : D^* \bullet Q)$. If $C \not\preceq D$ we have

$$\begin{array}{ll}
 & \mathcal{A}[\![(\exists z : D^* \bullet Q)[\mathsf{new}(C)/u]]\!](s,h)(s',h') \\
= & \{ \text{ def. } [\mathsf{new}(C)/u] \text{ and } D \not\preceq C \ \} \\
 & \mathcal{A}[\![(\exists z : D^* \bullet Q[\mathsf{new}(C)/u])]\!](s,h)(s',h') \\
= & \{ \text{ def. } \mathcal{A}[\![_]\!] \ \} \\
 & \mathcal{A}[\![Q[\mathsf{new}(C)/u]]\!](s[z\mapsto \bar v],h)(s',h') = tt \text{ for some } \bar v \text{ such that} \\
 & \bar v \in \mathit{dom}(D^*) \text{ and } \mathit{rng}(\bar v) \subseteq \mathit{dom}(h) \\
= & \{ \text{ ind. hyp. } \} \\
 & \mathcal{A}[\![Q]\!](s[z\mapsto \bar v][u\mapsto o], h\cdot[o\mapsto \mathit{init}(C)])(s',h') = tt \text{ for some } \bar v \text{ such} \\
 & \text{that } \bar v \in \mathit{dom}(D^*) \text{ and } \mathit{rng}(\bar v) \subseteq \mathit{dom}(h) \\
= & \{ \ C \not\preceq D \text{ and Lemma 2.2 imply } o \notin \mathit{dom}(D) \ \} \\
 & \mathcal{A}[\![Q]\!](s[z\mapsto \bar v][u\mapsto o], h\cdot[o\mapsto \mathit{init}(C)])(s',h') = tt \text{ for some } \bar v \text{ such} \\
 & \text{that } \bar v \in \mathit{dom}(D^*) \text{ and } \mathit{rng}(\bar v) \subseteq \mathit{dom}(h)\cup\{o\} \\
= & \{ \text{ def. } \mathcal{A}[\![_]\!] \text{ and } u \not\equiv z \ \} \\
 & \mathcal{A}[\![(\exists z : D^* \bullet Q)]\!](s[u\mapsto o], h\cdot[o\mapsto \mathit{init}(C)])(s',h') \ .
\end{array}$$

Now let $C \preceq D$. Then $(\exists z : D^* \bullet Q)[\mathsf{new}(C)/u]$ is syntactically equivalent to $(\exists z \bullet (\exists z' \bullet \mathsf{length}(z) = \mathsf{length}(z') \wedge (\mathsf{h}^*(Q\langle z\|^u_{z'}\rangle)[\mathsf{new}(C)/u])))$ by the definitions of h^* and $[\mathsf{new}(C)/u]$. Let us assume that the latter formula holds in the state (s,h) and the freeze state (s',h'). So there exists a sequence $\bar v$ such that $\bar v \in \mathit{dom}(D^*)$, $\mathit{rng}(\bar v) \subseteq \mathit{dom}(h)$, and a sequence of boolean values $\bar\beta \in \mathit{dom}(\mathsf{boolean}^*)$ of the same length such that

$$\mathcal{A}[\![\mathsf{h}^*(Q\langle z\|^u_{z'}\rangle)[\mathsf{new}(C)/u]]\!](s[z,z'\mapsto \bar v,\bar\beta],h)(s',h') = tt \ .$$

From the induction hypothesis follows that

$$\mathcal{A}[\![\mathsf{h}^*(Q\langle z\|^u_{z'}\rangle)]\!](s[z,z',u\mapsto \bar v,\bar\beta,o], h\cdot[o\mapsto \mathit{init}(C)])(s',h') = tt \ .$$

Then we get

$$\mathcal{A}[\![Q\langle z\|^u_{z'}\rangle]\!](s[z,z',u\mapsto \bar v,\bar\beta,o], h\cdot[o\mapsto \mathit{init}(C)])(s',h') = tt \ .$$

[1] We need a complexity measure that assigns a greater weight to the formula $(\exists z \bullet Q)$ than to $\mathsf{h}^*(Q\langle z\|^u_{z'}\rangle)$. Note that the operations h^* and $\langle z\|^u_{z'}\rangle$ can only increase the complexity of a formula. It therefore suffices to assign to every quantified formula $(\exists z \bullet Q)$ the standard complexity of $\mathsf{h}^*(Q\langle z\|^u_{z'}\rangle)$ plus some positive number.

using Lemma 6.4. Let $\bar{v}'$ be the sequence of objects of the same length as $\bar{v}$ such that the i-th element of $\bar{v}'$ is o if the i-th element of $\bar{\beta}$ is tt, and the i-th element of $\bar{v}$ otherwise, for every valid index of $\bar{v}'$. By Lemma 6.9 we then get

$$\mathcal{A}[\![Q]\!](s[u \mapsto o][z \mapsto \bar{v}'], h \cdot [o \mapsto \mathit{init}(C)])(s', h') = tt \ .$$

Note that $\bar{v}' \in \mathit{dom}(D^*)$ because $C \preceq D$ implies $o \in \mathit{dom}(D)$ (Lemma 2.2). Clearly, we also have $\mathit{rng}(\bar{v}') \subseteq \mathit{dom}(h) \cup \{o\}$. Therefore

$$\mathcal{A}[\![(\exists z : D^* \bullet Q)]\!](s[u \mapsto o], h \cdot [o \mapsto \mathit{init}(C)])(s', h') = tt \ , \tag{6.3}$$

which finishes one direction of the proof.

Our proof of the opposite direction starts from (6.3). Let $\bar{v}^*$ be a sequence such that $\bar{v}^* \in \mathit{dom}(D^*)$, $\mathit{rng}(\bar{v}^*) \subseteq \mathit{dom}(h) \cup \{o\}$, and

$$\mathcal{A}[\![Q]\!](s[u \mapsto o][z \mapsto \bar{v}^*], h \cdot [o \mapsto \mathit{init}(C)])(s', h') = tt \ .$$

Let $\bar{\beta}^*$ be the sequence of boolean values of the same length as $\bar{v}^*$ such that the i-th element of $\bar{\beta}$ is tt if and only if o is the i-th element of $\bar{v}^*$, for every valid index i of $\bar{\beta}$. Next, consider the sequence $\bar{v}^\dagger$ that is obtained from $\bar{v}^*$ by replacing every occurrence of o by *null*. Clearly, we have $\bar{v}^\dagger \in \mathit{dom}(D^*)$ and $\bar{v}^* \sqsubseteq \mathit{dom}(h)$.

At this point, we use Lemma 6.9 in the opposite direction to obtain

$$\mathcal{A}[\![Q\langle z \| ^u_{z'} \rangle]\!](s[u \mapsto o][z, z' \mapsto \bar{v}^\dagger, \bar{\beta}^*], h \cdot [o \mapsto \mathit{init}(C)])(s', h') = tt \ ,$$

which in turn implies that

$$\mathcal{A}[\![\mathsf{h}^*(Q\langle z \| ^u_{z'} \rangle)]\!](s[u \mapsto o][z, z' \mapsto \bar{v}^\dagger, \bar{\beta}^*], h \cdot [o \mapsto \mathit{init}(C)])(s', h') = tt \ ,$$

holds according to Lemma 6.4. And by the induction hypothesis we then get

$$\mathcal{A}[\![\mathsf{h}^*(Q\langle z \| ^u_{z'} \rangle)[\mathsf{new}(C)/u]]\!](s[z, z' \mapsto \bar{v}^\dagger, \bar{\beta}^*], h)(s', h') = tt \ . \tag{6.4}$$

Recall that $((\exists z : D^* \bullet Q)[\mathsf{new}(C)/u])$ is the formula

$$(\exists z \bullet (\exists z' \bullet \mathsf{length}(z) = \mathsf{length}(z') \wedge (\mathsf{h}^*(Q\langle z \| ^u_{z'} \rangle)[\mathsf{new}(C)/u]))) \ .$$

It is not difficult to prove that $\mathcal{A}[\![(\exists z : D^* \bullet Q)[\mathsf{new}(C)/u]]\!](s, h)(s', h') = tt$ follows from (6.4) by the definition of $\mathcal{A}[\![\_]\!]$ and the construction of $\bar{v}^\dagger$ and $\bar{\beta}^*$. □

Our first theorem in this chapter states that $\mathsf{h}^*(Q)[\mathsf{new}(C)/u]$ is a valid precondition of Q with respect to $u := \mathsf{alloc}(C)$.

Theorem 6.11. *For every assertion Q, and every class C and local variable u such that $C \preceq [u]$, we have* $\models \{\mathsf{h}^*(Q)[\mathsf{new}(C)/u]\}\ u := \mathsf{alloc}(C)\ \{Q\}$.

Proof. Let $\langle u := \mathsf{alloc}(C), (s,h)\rangle \to (s'', h'')$ and $(s,h)(s',h') \models \mathsf{h}^*(Q)[\mathsf{new}(C)/u]$. This computation can only be derived using rule OA of the operation semantics. Hence the assumption of that rule, which says that there exists and object $o = (C,i)$ such that $h(o)$ is undefined, also holds. Moreover, we have $s'' = s[u \mapsto o]$ and $h'' = h \cdot [o \mapsto \mathit{init}(C)]$. By Lemma 6.3 and Lemma 6.10 we get $(s'', h'')(s', h') \models \mathsf{h}^*Q$. Then $(s'', h'')(s', h') \models Q$ follows from Lemma 6.4. □

Our second theorem of this chapter shows that $\mathsf{h}^*(Q)[\mathsf{new}(C)/u]$ is not only a valid precondition for every postcondition Q, but that it is also the *weakest* possible precondition of $u := \mathsf{alloc}(C)$ with respect to Q. Moreover, it entails that the validity of the assertion $Q \to (\mathsf{h}^*(Q')[\mathsf{new}(C)/u])$ implies the validity of the Hoare triple $\{Q\}\ u := e\ \{Q'\}$ for every pair of assertions Q and Q', which means that we can show the validity of a proof outline $\models \{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\}$ by proving the formula $Q \to (\mathsf{h}^*(Q')[\mathsf{new}(C)/u])$.

Theorem 6.12. *Let Q, Q' be arbitrary assertions. Let C be a class , and let u be a local variable such that $C \preceq [u]$. Then*

$$\models \{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\} \quad \Longleftrightarrow \quad \models Q \to (\mathsf{h}^*(Q')[\mathsf{new}(C)/u])\ .$$

Proof. First, we prove that the validity of $\{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\}$ implies $\models Q \to (\mathsf{h}^*(Q')[\mathsf{new}(C)/u])$. Let $(s,h)(s',h') \models Q$, and let $o = (C,i)$ be such that $h(o)$ is undefined. Then we can derive (using rule OA) the computation $\langle u := \mathsf{alloc}(C), (s,h)\rangle \to (s'', h'')$, where $s'' = s[u \mapsto o]$, and the heap $h'' = h \cdot [o \mapsto \mathit{init}(C)]$. From $\models \{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\}$ and $(s,h)(s',h') \models Q$ we may then conclude that $(s'', h'')(s', h') \models Q'$, which according to Lemma 6.4 and Lemma 6.10 is equivalent to $(s,h)(s',h') \models \mathsf{h}^*(Q')[\mathsf{new}(C)/u]$.

To prove the remaining implication we assume that $(s,h)(s',h') \models Q$, and that $\langle u := \mathsf{alloc}(C), (s,h)\rangle \to (s'', h'')$. By the validity of $Q \to (\mathsf{h}^*(Q')[\mathsf{new}(C)/u])$ we get $(s,h)(s',h') \models (\mathsf{h}^*(Q')[\mathsf{new}(C)/u])$. The assumption of rule OA, which must have been used to derive the above computation, states that there exists an $o = (C,i)$ such that $h(o)$ is undefined. Moreover, the rule ensures that $s'' = s[u \mapsto o]$ and that $h'' = h \cdot [o \mapsto \mathit{init}(C)]$. Hence Lemma 6.10 implies that $(s'', h'')(s', h') \models \mathsf{h}^*(Q')$. Then Lemma 6.4 yields $(s'', h'')(s', h') \models Q'$. □

6.1.2 Strongest Postconditions of Object Allocation

We have seen in the previous section how one can reason about object allocation using a weakest precondition calculus. We will show in this section that it is also possible to reason about object allocation 'in the opposite direction', i.e., by means of the strongest postcondition of object allocation. We will construct a formula in our assertion language that expresses the strongest postcondition $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$ of the statement $u := \mathsf{alloc}(C)$ with respect to a precondition Q.

There are four aspects of object allocation that need to be reflected by the strongest postcondition $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$. Firstly, it must describe the

internal state of the new object. Secondly, it has to fix the allocated type of the new object. Thirdly, it must express the 'freshness' of the new object. That is, it should imply that u is the only reference to the new object after the allocation of the new object. And finally, it must state that the properties of the state as described by Q still hold (although they do not apply to Q). This assumption is valid because object allocation has no effect on existing parts of the state.

The first aspect concerns the fields of the new object. Every field of the new object has its default value after allocation, which is expressed by the following formula.

$$\neg(u = \mathsf{null}) \wedge \bigwedge\nolimits_{D \in SuperCls(C)} \left(\bigwedge\nolimits_{x \in Fields(D)} \left(((D)u).x = \mathsf{init}([x]) \right) \right) \qquad (Q_1)$$

Here $SuperCls(C)$ denotes the set $\{D | C \trianglelefteq D\}$, i.e., the set that contains every superclass of C.

The second aspect is even simpler to describe. The formula (Q_2) below states that the new object u belong to the domain of C, and that it does not belong to the domain of some subclass of D, which implies that its dynamic type is C:

$$u \text{ instanceof } C \wedge \bigwedge\nolimits_{D \in PrpSubTps(C)} \neg u \text{ instanceof } D \ . \qquad (Q_2)$$

The expressions $PrpSubTps(C)$ in this formula denotes the set $\{D | D \prec C\}$, which contains all proper subtypes of C. (An obvious optimization would be to consider only the immediate subtypes of C, but we use the present formula for simplicity.)

We now turn out attention to the third aspect: the 'freshness' of the new object. The identity of the new object is unequal to the identity of any of the previously allocated object. We express this fact using a logical variable $\mathcal{O} : C^*$ which is assumed to contain all other objects that also belong to $dom(C)$ (we will eventually existentially quantify over this variable). Our third clause says that u is the only object that does not occur in this sequence $\mathcal{O}$.

$$\neg(u \in \mathcal{O}) \wedge (\forall o : C \bullet o = u \vee o \in \mathcal{O}) \qquad (Q_3)$$

The freshness of the identity of the new object does also imply that no other variable stores a reference to the new object. More precisely, no instance variable, logical variable or local variable other than u may point to the new object. The following formula says that no field of any object references the new object.

$$\bigwedge\nolimits_C \left(\bigwedge\nolimits_{x \in Fields(C) \cap \{x' | C \preceq [x']\}} (\forall o : C \bullet \neg(o.x = u)) \right) \qquad (Q_4)$$

It is also not difficult to express that every local variable different from u does not point to the new object; only a finite number of local variables are in scope, and we assume in (Q_5) that v is one of those variables.

$$\bigwedge\nolimits_{v \in \{v_0 | C \preceq [v_0]\} \setminus \{u\}} \neg(v = u) \qquad (Q_5)$$

However, the set of logical variables that may occur in proof outlines is not *a priori* finite. This makes it impossible to express in a finite first-order formula that none of these variables references the new object. We therefore only say in (Q_6) below that the logical variables that occur free in the precondition Q do not point to the new object. This decision has consequences for the set of Hoare triples that we can allow. We will say more about this issue below.

$$\bigwedge\nolimits_{z \in FLV(Q) \cap \{z_0 \mid C \preceq [z_0]\}} \neg(z = u) \wedge \bigwedge\nolimits_{z \in FLV(Q) \cap \{z_0 \mid [z_0] = D^* \text{ and } C \preceq D\}} \neg(u \in z) \qquad (Q_6)$$

We assume here that $FLV(Q)$ denotes the set of logical variables that occur free in Q.

Finally, we should also say that the new object does not exist in the freeze state. More precisely, we should state that every expression of the form $\mathsf{old}(.)$ does not denote the new object. Again, this is problematic because there are infinitely many expressions of this form. We solve this issue similarly by only taking expressions of the form $\mathsf{old}(e)$ into account of which e is a subexpression of some expression e' such that $\mathsf{old}(e')$ occurs in the precondition Q.

$$\bigwedge\nolimits_{e \in \{e_0 \mid e_0 \in SubExprs(e_1) \text{ and } C \preceq [e_0] \text{ and } \mathsf{old}(e_1) \in SubExprs(Q)\}} \neg(\mathsf{old}(e) = u) \qquad (Q_7)$$

We assume that $SubExprs(Q)$ denotes the set of all subexpressions of Q.

The final aspect of the strongest postcondition concerns the properties of the initial state as described by the precondition Q. As observed above, these properties still hold after object allocation because the only existing part of the state that is being modified by the addition of a new object is the local variable u. We will therefore replace this variable by a existentially quantified logical variable z.

Furthermore, we must ensure that the meaning of quantified subformulas in Q is restricted to objects of the initial state. This can be achieved by restricting quantification in Q to the objects in Q. We denote a formula Q in which quantification is restricted to $\mathcal{O}$ by $Q \downarrow_{\mathcal{O}}^{C}$. Formally, we define $Q \downarrow_{\mathcal{O}}^{C}$ by structural induction on Q. The only interesting cases are as follows.

$$\begin{array}{rcl}
(\exists z : t \bullet Q) \downarrow_{\mathcal{O}}^{C} & \equiv & (\exists z : t \bullet (Q \downarrow_{\mathcal{O}}^{C})) \quad \text{for } t \in \{\mathsf{boolean}, \mathsf{int}\} \\
(\exists z : t^* \bullet Q) \downarrow_{\mathcal{O}}^{C} & \equiv & (\exists z : t \bullet (Q \downarrow_{\mathcal{O}}^{C})) \quad \text{for } t \in \{\mathsf{boolean}, \mathsf{int}\} \\
(\exists z : D \bullet Q') \downarrow_{\mathcal{O}}^{C} & \equiv & \begin{cases} (\exists z : D \bullet z \in \mathcal{O} \wedge (Q \downarrow_{\mathcal{O}}^{C})) & \text{if } C \preceq D \\ (\exists z : D \bullet (Q \downarrow_{\mathcal{O}}^{C})) & \text{otherwise} \end{cases} \\
(\exists z : D \bullet Q') \downarrow_{\mathcal{O}}^{C} & \equiv & \begin{cases} (\exists z : D \bullet z \in \mathcal{O} \wedge (Q \downarrow_{\mathcal{O}}^{C})) & \text{if } C \preceq D \\ (\exists z : D \bullet (Q \downarrow_{\mathcal{O}}^{C})) & \text{otherwise} \end{cases}
\end{array}$$

The final clause of our strongest postcondition of a statement $u := \mathsf{alloc}(C)$ with respect to an arbitrary precondition Q is then the formula

$$(\exists z : [u] \bullet \neg(z = u) \wedge (Q[z/u]) \downarrow_{\mathcal{O}}^{C}) \ . \qquad (Q_8)$$

The following lemma basically says that Q holds before object allocation if and only if Q_8 holds after the object allocation provided that $\mathcal{O}$ denotes a sequence of all the objects in $dom(C)$ that existed in the initial state.

Lemma 6.13. *Let Q be an arbitrary assertion, and let C be an arbitrary class. Let (s,h) be an arbitrary state, and let (s',h') be a compatible freeze state. Let $o=(C,i)$ be such that $h(o)$ is undefined. Let u be a local variable such that $C \preceq [u]$. Assume that $\mathcal{O} : C^*$ and $z : [u]$ are two logical variables that do not occur in Q. Let $\bar{v}$ be a sequence of the objects in $dom(h) \cap dom(C)$. Then*

$$\mathcal{A}[\![Q]\!](s,h)(s',h') = \mathcal{A}[\![Q[z/u] \downarrow^C_{\mathcal{O}}]\!](s'',h'')(s',h') \ ,$$

where $s'' = s[z, \mathcal{O} \mapsto s(u), \bar{v}]$ and $h'' = h \cdot [o \mapsto init(C)]$.

Proof. By structural induction on Q. □

With clauses Q_1-Q_8 we can define $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$. It says that there exists a sequence of C-objects such that the clauses Q_1 up to Q_8 hold.

$$\mathsf{sp}(Q, u := \mathsf{alloc}(C)) \equiv (\exists \mathcal{O} : C^* \bullet \bigwedge\nolimits_{i=1}^{8} Q_i) \ .$$

In the remainder of this section, we will discuss two properties of the formula $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$. Firstly, we will prove that $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$ is a *valid* postcondition of the precondition Q for the statement $u := \mathsf{alloc}(C)$.

Lemma 6.14. *For every precondition Q, every local variable u, and every class C we have $\models \{Q\}\ u := \mathsf{alloc}(C)\ \{\mathsf{sp}(Q, u := \mathsf{alloc}(C))\}$.*

Proof. Straightforward. The strongest postcondition holds in the final state if we assign a sequence of the objects in $dom(C)$ that existed in the initial state to $\mathcal{O}$. The only interesting part of the proof, which concerns formula Q_8, is described by Lemma 6.13. □

Secondly, we would like to prove that

$$\models \{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\} \text{ implies } \models \mathsf{sp}(Q, u := \mathsf{alloc}(C)) \rightarrow Q' \ ,$$

for every postcondition Q'. This property says that $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$ is the *strongest* postcondition of $u := \mathsf{alloc}(C)$ with respect to the precondition Q.

Unfortunately, the latter property does not hold for arbitrary Hoare triples $\{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\}$. This is ultimately caused by the fact that we cannot express the freshness of an object with respect to the values of an *infinite* set of variables by means of a *finite* formula. Hence we had to limit quantification in (Q_6) to logical variables that occur free in Q. Similarly, we restricted quantification in (Q_7) to expressions of the form $\mathsf{old}(e)$ that occur in Q.

We address this issue by putting some additional restrictions on Hoare triples. We will prove that the property holds for all Hoare triples that satisfy these restrictions.

Definition 6.1. *A Hoare triple $\{Q\}\ S\ \{Q'\}$ (see Section 3.2) is conservative if*

- *every logical variable that occurs free in Q' also occurs free in Q, and*

- *for every subexpression of Q' of the form $\mathsf{old}(e)$ there exists an expression e' such that e is a subexpression of e' and $\mathsf{old}(e')$ is a subexpression of Q.*

This restriction seems quite natural because the purpose behind the use of logical variables is to freeze initial values of program variables. Logical variables can only fulfil this task if they occur in the precondition of the Hoare triple. The same can be said about expressions of the form $\mathsf{old}(e)$. We will *not* restrict Hoare triples of the form $\{P\}\ m@C\ \{Q\}$ in the same way because expressions of the form $\mathsf{old}(e)$ are not allowed in P.

We need the following additional definitions in the sequel of our investigation of the properties of $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$. Let $h \setminus \{o\}$ denote the heap that is obtained from h by removing object o from its domain, and by assigning *null* to every field that points to o. Let $s \setminus \{o\}$ be the stack that is obtained from s by assigning *null* to every local and logical variable v with $s(v) = o$, and by replacing o by *null* in every sequence of values $s(z)$ of a sequence variable z.

We will present a series of lemmas that describe the effect of object removals on the validity of assertions. Our first lemma states that the removal of an object from the heap has no effect on formulas in which quantification is restricted.

Lemma 6.15. *Let Q be an arbitrary assertion, and let u be a local variable such that $C \preceq [u]$. Let (s, h) be a state such that $s(u) = o$ and $o \in dom(h) \cap dom(C)$, and s is consistent with $h \setminus \{o\}$. Let (s', h') be a freeze state that is compatible with $(s, h \setminus \{o\})$. Then*

$$\mathcal{A}[\![Q \downarrow^C_{\mathcal{O}}]\!](s, h)(s', h') = \mathcal{A}[\![Q \downarrow^C_{\mathcal{O}}]\!](s, h \setminus \{o\})(s', h') \ .$$

Proof. By structural induction on Q. □

Our second lemma says that we can remove an object from the stack without changing the validity of an assertion if the assertion has no free occurrences of variables that reference the removed object before the removal. However, the lemma also supports one specific reference to the old object that is restored.

Lemma 6.16. *Let Q be an arbitrary assertion. For every state (s, h) and compatible freeze state (s', h'), and every object $o \in dom(h) \cap dom(C)$ such that $s(u) = o$, we have that if*

$$(s, h)(s', h') \models \bigwedge\nolimits_{v \in \{v_0 \mid C \preceq [v_0]\} \setminus \{u\}} \neg(v = u) \ ,$$

$$(s, h)(s', h') \models \bigwedge\nolimits_{z \in FLV(Q) \cap \{z_0 \mid C \preceq [z_0]\}} \neg(z = u) \ , \ \textit{and}$$

$$(s, h)(s', h') \models \bigwedge\nolimits_{z \in FLV(Q) \cap \{z_0 \mid [z_0] = D^* \ \textit{and}\ C \preceq D\}} \neg(u \in z)$$

then

$$\mathcal{A}[\![Q]\!](s, h)(s', h') = \mathcal{A}[\![Q]\!]((s \setminus \{o\})[u \mapsto o], h)(s', h') \ .$$

Proof. By structural induction on Q. □

Our final lemma describes the effect of object removal in the freeze state.

Lemma 6.17. *Let Q be an arbitrary assertion. For every state (s,h) and compatible freeze state (s',h'), and every object $o \in dom(h) \cap dom(C)$ such that $s(u) = o$, we have that if*

$$\bigwedge_{e \in \{e_0 \mid e_0 \in SubExprs(e_1) \text{ and } C \preceq [e_0] \text{ and } \mathsf{old}(e_1) \in SubExprs(Q)\}} \neg(\mathsf{old}(e) = u)$$

holds in the state (s,h) and the freeze state (s',h') then

$$\mathcal{A}[\![Q]\!](s,h)(s',h') = \mathcal{A}[\![Q]\!](s,h)(s' \setminus \{o\}, h' \setminus \{o\}) \ .$$

Proof. By structural induction on Q. □

Our final result in this section is a theorem that states that the formula $\mathsf{sp}(Q, u := \mathsf{alloc}(C))$ as defined above denotes the strongest postcondition of $u := \mathsf{alloc}(C)$ with respect to Q.

Theorem 6.18. *Let C be an arbitrary class, and let u be an arbitrary local variable. Let $\{Q\}\ u := \mathsf{alloc}(C)\ \{Q\}$ be a conservative Hoare triple. Then*

$$\models \{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\} \text{ implies } \models \mathsf{sp}(Q, u := \mathsf{alloc}(C)) \rightarrow Q' \ .$$

Proof. Assume that $(s,h)(s',h') \models \mathsf{sp}(Q, u := \mathsf{alloc}(C))$. This means that there exists a sequence $\bar{v}$ of objects from $dom(h) \cap dom(C)$ such that

$$(s[\mathcal{O} \mapsto \bar{v}], h)(s',h') \models \bigwedge_{i=1}^{8} SP_i \ . \tag{6.5}$$

Let $o = s(u)$. We will prove that there exists a state (s_0,h_0) and a compatible freeze state (s_0',h_0') such that

$$\langle u := \mathsf{alloc}(C), (s_0,h_0)\rangle \rightarrow (s_0'',h_0'') \text{ and } (s_0,h_0)(s_0',h_0') \models Q \ , \tag{6.6}$$

where $s_0'' = s_0[u \mapsto o]$, and $h_0'' = h_0 \cdot [o \mapsto init(C)]$. By the validity of the Hoare triple $\{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\}$ and (6.9) this implies

$$(s_0'',h_0'')(s_0',h_0') \models Q' \ . \tag{6.7}$$

Finally, we will argue that (6.7) implies that $(s,h)(s',h') \models Q'$.

We start with the state (s_0,h_0). Observe that (6.5) states that we have $(s[\mathcal{O} \mapsto \bar{v}], h)(s',h') \models Q_8$. Hence there exists a value $v_0 \in dom([u]) \cap dom(h)$ such that $v_0 \neq o$, and moreover

$$(s[\mathcal{O}, z \mapsto \bar{v}, v_0], h)(s',h') \models Q[z/u]\downarrow_{\mathcal{O}}^{C} \ . \tag{6.8}$$

This value v_0 will be the value of u in s_0. Let $s_0 = (s \setminus \{o\})[u \mapsto v_0]$ and $h_0 = h \setminus \{o\}$. Thus s_0 is consistent with h_0.

The state (s_0, h_0) satisfies the premiss of rule (OA) because $h_0(o)$ is undefined and because Q_2 in (6.5) implies that the dynamic type of o is C. Hence

$$\langle u := \mathsf{alloc}(C), (s_0, h_0)\rangle \to (s_0'', h_0'') \ . \tag{6.9}$$

Next, let $s_0' = s' \setminus \{o\}$ and $h_0' = h' \setminus \{o\}$. Our new aim is to prove that $(s_0, h_0)(s_0', h_0') \models Q$. According to Lemma 6.13 it suffices to prove

$$(s_0[\mathcal{O}, z \mapsto \bar{v}, v_0], h_0)(s_0', h_0') \models Q[z/u] \downarrow_{\mathcal{O}}^{C} \ . \tag{6.10}$$

(Note that Q_3 in (6.5) is needed here to ensure that $\bar{v}$ is a sequence of the objects in $dom(h_0) \cap dom(C)$).) Lemma 6.15 states that Eq. (6.10) is equivalent to

$$(s_0[\mathcal{O}, z \mapsto \bar{v}, v_0], h)(s_0', h_0') \models Q[z/u] \downarrow_{\mathcal{O}}^{C} \tag{6.11}$$

because the clause $\neg(u \in \mathcal{O})$ of Q_3 in (6.5) ensures that $s_0[\mathcal{O} \mapsto \bar{v}]$ is consistent with h_0, and moreover $v_0 \neq o$. Observe that

$$s_0[\mathcal{O}, z \mapsto \bar{v}, v_0] = (s \setminus \{o\})[u \mapsto v_0][\mathcal{O}, z \mapsto \bar{v}, v_0] = (s[u, \mathcal{O}, z \mapsto v_0, \bar{v}, v_0]) \setminus \{o\}$$

because $z_0 \neq o$ and $o \notin \bar{v}$ (Q_3 in (6.5)). Then we can use Lemma 6.16 to prove that (6.11) is equivalent to

$$(s[u, \mathcal{O}, z \mapsto v_0, \bar{v}, v_0], h)(s_0', h_0') \models Q[z/u] \downarrow_{\mathcal{O}}^{C} \ , \tag{6.12}$$

which in turn is equivalent to

$$(s[\mathcal{O}, z \mapsto \bar{v}, v_0], h)(s_0', h_0') \models Q[z/u] \downarrow_{\mathcal{O}}^{C} \tag{6.13}$$

because u does not occur in $Q[z/u] \downarrow_{\mathcal{O}}^{C}$. Finally, we observe that (6.13) follows from (6.8) via Lemma 6.17 because clause Q_7 in (6.5) ensures that the premisses of Lemma 6.17 hold. This line of reasoning therefore shows that $(s_0, h_0)(s_0', h_0') \models Q$.

Using the validity of $\{Q\}\ u := \mathsf{alloc}(C)\ \{Q'\}$ and the computation in (6.9) we then conclude

$$(s_0'', h_0'')(s_0', h_0') \models Q' \ . \tag{6.14}$$

Recall that $h_0'' = h_0 \cdot [o \mapsto init(C)] = (h \setminus \{o\}) \cdot [o \mapsto init(C)]$. Note that Q_4 in (6.5) implies that h_0 assigns the same values as h to fields of objects other than o because not a single instance variable points to o in h. We claim that $(h \setminus \{o\}) \cdot [o \mapsto init(C)] = h$ follows from (6.5) because Q_1 says that each instance variable of o has its default value in h. Hence (6.14) is equivalent to

$$(s_0'', h)(s_0', h_0') \models Q' \ . \tag{6.15}$$

Recall that $s_0'' = s_0[u \mapsto o] = s(u \setminus \{o\})[u \mapsto v_0][u \mapsto o] = s(u \setminus \{o\})[u \mapsto o]$. Because Q' is part of a conservative Hoare triple we know that every logical variable that occurs free in Q' also occurs in Q. For this reason, we have by (6.5) (in particular Q_6) that each logical variable that occurs free in Q' does not point to o. Similarly, we have by Q_5 that each local variable v that is distinct from u does not reference the new object. Therefore we can use Lemma 6.16 to obtain $(s, h)(s_0', h_0') \models Q'$ from (6.15). Observe that Q_7 in (6.5) implies that the conditions of Lemma 6.17 are met. Hence $(s, h)(s', h') \models Q'$. □

6.2 Object Initialization

As explained in the introduction of this chapter, object allocation is only the first part of the execution of a statement $u := \mathsf{new}\ C(e_1, \ldots, e_n)$. Upon completion of the allocation, the constructor method of class C is executed with the new object as receiver. A constructor method typically initializes the fields of a new objects with specific values. Finally, a reference to the new object is assigned to the local variable u. (Rule OC in Section 2.2.3 provides a formal description of the execution of $u := \mathsf{new}\ C(e_1, \ldots, e_n)$.)

In this section, we will define the verification condition of a Hoare triple of the form $\{Q\}\ u := \mathsf{new}\ C(e_1, \ldots, e_n)\ \{Q'\}$. Interestingly, such a verification condition combines three previous results. We will employ our analysis of object allocation in the previous section to reason about the first phase, showing that we can either use the weakest precondition calculus from Section 6.1.1 or the strongest postcondition from Section 6.1.2. Reasoning about the call to the constructor method will be done using our techniques for reasoning about method calls as described in Section 5.2.2. The final assignment to the local variable u can be handled using our weakest precondition calculus for assignments to local variables, as described in Section 4.1. Our reuse of these parts in this section shows that we have developed some fairly general techniques in the previous two chapters, which can function as building blocks in the axiomatization of more complex statements.

Our description of the execution of a statement $u := \mathsf{new}\ C(e_1, \ldots, e_n)$ reveals that such a statement is equivalent to the execution of the following sequence of more basic statements.

$$v := \mathsf{alloc}(C)\ ;\ v.C(e_1, \ldots, e_n)\ ;\ u := v$$

We assume here that v is a fresh local variable, and that $v.C(e_1, \ldots, e_n)$ denotes a call of the constructor method in class C with receiver v and actual parameters $e_1, \ldots, e_n$. We first assign the new object to v instead of u because the local variable u may occur in the parameters $e_1, \ldots, e_n$, and u should still have its original value while these expressions are evaluated.

We are looking for a verification condition that tells us if a Hoare triple $\{Q\}\ u := \mathsf{new}\ C(e_1, \ldots, e_n)\{Q'\}$ is valid. Our analysis suggests that we can also try to find a verification condition for the Hoare triple

$$\{Q\}\ v := \mathsf{alloc}(C)\ ;\ v.C(e_1, \ldots, e_n)\ ;\ u := v\ \{Q'\}\ .$$

However, this Hoare triple has two annotation points (semicolons) that are not described by assertions. So the question is which assertions correctly describe the state at each of these two annotation points.

Possible answers to these two questions are as follows. We have shown in the previous section that the formula $\mathsf{sp}(Q, v := \mathsf{alloc}(C))$ leads to a valid Hoare triple $\{Q\}\ v := \mathsf{alloc}(C)\ \{\mathsf{sp}(Q, v := \mathsf{alloc}(C))\}$ (cf. Lemma 6.14). So $\mathsf{sp}(Q, v := \mathsf{alloc}(C))$ could be used to describe the state at the first annotation

point. Similarly, we can use the weakest precondition of $u := v$ with respect to Q' to describe the state at the second annotation point. Theorem 4.4 implies that $\{Q'[v/u]\}\ u := v\ \{Q'\}$ is a valid Hoare triple (the predicate $\mathsf{defined}(v)$ is equivalent to true and can therefore be omitted). These observations imply that we can use the verification condition of the Hoare triple

$$\{\mathsf{sp}(Q, v := \mathsf{alloc}(C))\}\ v.C(e_1, \ldots, e_n)\ \{Q'[v/u]\} \tag{6.16}$$

to check the validity of $\{Q\}\ u := \mathsf{new}\ C(e_1, \ldots, e_n)\ \{Q'\}$.

A verification condition for a call to a constructor method is less complex than a verification condition for a dynamically bound method call (VC_j on p. 84) because a call to a constructor method can be bound statically to a method implementation: a call $v.C(e_1, \ldots, e_n)$ always triggers the constructor method in class C. We can use the specification of the constructor method in class C to justify such a call along the lines of our analysis of statically bound method calls in Section 5.2.2.

We will assume that constructor methods are annotated in the same way as ordinary methods. That is, each constructor method has a precondition and a postcondition. Let P_{cnst} be the precondition of the constructor method in class C, and let Q_{cnst} be its postcondition. We are looking for a verification condition VCC whose validity implies that the Hoare triple in (6.16) is valid, provided that the specification of the constructor method is also valid. The starting point of our verification condition VCC will be the verification condition $VC5$ (see p. 80) for statically bound method calls. We will argue below that a slightly simpler verification condition suffices to reason about calls to constructor methods. The resulting formula is as follows.

$$\begin{aligned}&\lfloor(\textstyle\bigwedge_{i=1}^{n} \mathsf{defined}(e_i))\rfloor \wedge \mathsf{heap}_V \wedge \lfloor \mathsf{sp}(Q, v := \mathsf{alloc}(C))\rfloor \wedge \\ &\qquad (\forall \bar{z} \in \mathcal{H} \bullet \mathsf{g}(P_{cnst}) \rightarrow (Q_{cnst}[v/\mathsf{this}][\mathsf{g}(.)/\mathsf{old}(.)]) \rightarrow Q'[v/u] \quad (VCC)\end{aligned}$$

Our first change to the original verification is the omission of the clauses $\mathsf{defined}(v)$ and $\neg(v = \mathsf{null})$ ($\mathsf{defined}(e_0)$ and $\neg(e_0 = \mathsf{null})$ in $VC5$). The first clause is redundant because $\mathsf{defined}(v)$ is equivalent to true. The second clause is also redundant because $\neg(v = \mathsf{null})$ already follows from $\mathsf{sp}(Q, v := \mathsf{alloc}(C))$. The formula heap_V is defined in the same way as in $VC5$. We also restrict the meaning of the precondition of the method call (the formula $\mathsf{sp}(Q, v := \mathsf{alloc}(C))$ in our present setting) again to the dual heap.

Another simplification is the replacement of the logical variable rec by v, which resulted in the substitution $[v/\mathsf{this}]$, and the omission of the corresponding clause $\mathsf{this} = \mathsf{rec}$. This change is possible because the expression in the call that denotes its receiver is the local variable v. The value of v does not change during the call, which enables it to fulfil the role of the logical variable rec. In the original rule we allow arbitrary expressions e_0 at the position of v in the call. The value of an expression e_0 may change during a call if it depends on the values of object fields.

The function symbol g has the same role in *VCC* as in *VC5*. It denotes the syntactical function that takes a predicate P and returns the predicate $\lfloor P[v, \bar{e}/\mathsf{this}, \bar{p}] \rfloor$. We assume again that $\bar{e} = e_1, \ldots, e_n$ and $\bar{p} = p_1, \ldots, p_n$.

Our final simplification is the exclusion of the substitution $[\mathsf{result}/u]$. This operations models the assignment of the result value to a local variable u in *VC5*. A constructor method, however, does not return a value.

Summarizing, we get the following rule for object creation.

$$\frac{\{P_{cnst}\}\ C@C\ \{Q_{cnst}\} \qquad VCC}{\{Q\}\ u := \mathsf{new}\ C(e_1, \ldots, e_n)\ \{Q'\}} \tag{6.17}$$

Here $\{P_{cnst}\}\ C@C\ \{Q_{cnst}\}$ denotes a Hoare triple that specifies the behavior of the constructor method of class C (the first occurrence of C in $C@C$ denotes the identifier of the constructor method, whereas the second occurrence reflects the class in which the method is defined). The validity of an interface specification of the form $\{P\}\ C@C\ \{Q\}$ follows from the validity of the set of proof obligations of its body as defined in Definition 3.5. We will further explain this issue in Section 7.2.

Theorem 6.19. *Rule 6.17 is sound.*

Proof. The proof of this theorem is a straightforward combination of the proofs of Theorem 4.4, Theorem 5.11 and Theorem 6.18. □

An advantage of rule 6.17 is that it automatically computes assertions for the two unspecified annotation points. The assertion that is used at the first annotation point is the strongest postcondition of the allocation of a new object with respect to the precondition. This solution, however, also has a distinct downside. The strongest postcondition is a rather long and unwieldy formula, as we have seen in the previous section. This implies that a verification condition that is based on this formula also becomes difficult to handle. Moreover, it may well be the case that most of the information in the strongest postcondition is superfluous. It is not necessary, for example, to include the information that a particular field of the new object has its default value after allocation if that value is discarded as a result of a field update before it is used.

This disadvantage of rule 6.17 was the reason behind our decision to describe an alternative approach to reasoning about object creation elsewhere [PdB05b]. We will also present this solution here. It is based on the weakest precondition calculus of object allocation in Section 6.1.1.

The idea behind our second approach is to allow a programmer to specify an additional assertion which describes the state after the allocation of the new object. This additional assertion is preceded by the keyword intermediate to distinguish it from the ordinary precondition of a statement $u := \mathsf{new}\ C(e_1, \ldots, e_n)$. This involves a change to the annotation schema outlined in Section 3.3. A sufficiently annotated statement of this form would be as follows.

$$\mathsf{assert}\ Q;\ \mathsf{intermediate}\ Q';\ u := \mathsf{new}\ C(e_1, \ldots, e_n)\ ;\ \mathsf{assert}\ Q'';$$

Note that there is no local variable is this approach that denotes the new object in the state described by Q'. For this purpose, we introduce a special-purpose local variable fresh that designates the new object in Q'. That is, we assume that the allocation step is modelled by the statement $\mathsf{fresh} := \mathsf{alloc}(C)$. The keyword fresh may only occur in Q'.

The annotated code fragment above has two verifications. There is one verification condition that checks whether $\{Q'\}\mathsf{fresh} := \mathsf{alloc}(C)\{Q'\}$ is a valid proof outline, and another verification condition that checks whether the remaining part is valid. The first verification condition is simply

$$Q \rightarrow \mathsf{h}^*(Q')[\mathsf{new}(C)/\mathsf{fresh}] \ . \tag{6.18}$$

In other words, we check whether Q implies the weakest precondition of the statement $\mathsf{fresh} := \mathsf{alloc}(C)$ with respect to Q'. The other verification condition is a variant of *VCC*:

$$\begin{aligned} &\lfloor(\bigwedge\nolimits_{i=1}^{n} \mathsf{defined}(e_i))\rfloor \wedge \neg(\mathsf{fresh} = \mathsf{null}) \wedge \mathsf{heap}_V \wedge \lfloor Q'\rfloor \wedge \\ &\quad (\forall \bar{z} \in \mathcal{H} \bullet \mathsf{g}(P_{cnst}) \rightarrow (Q_{cnst}[\mathsf{fresh}/\mathsf{this}][\mathsf{g}(.)/\mathsf{old}(.)]) \rightarrow Q''[\mathsf{fresh}/u] \end{aligned} \tag{6.19}$$

We assume here again that P_{cnst} and Q_{cnst} are the precondition and the postcondition of the constructor method in class C, respectively. All other elements of (6.19) are defined in the same way as in *VCC*.

We believe that this second approach to reasoning about object creation is more efficient in most cases because the weakest precondition operation in (6.19) leads to a more concise formula than the strongest postcondition operation in *VCC*. Moreover, both approaches require an equal number of annotations because the formula $\mathsf{h}^*(Q')[\mathsf{new}(C)/\mathsf{fresh}]$ - which can be computed automatically - can be used instead of Q in the proof outline. This choice would also mean that (6.18) becomes a tautology.

6.2.1 An Example

We will finish our discussion of object creation with an example that involves a constructor method. The example concerns a simple class *Clonable* with only two methods: a constructor method and a *clone* method. A complete proof outline of the class is listed in Figure 6.2.

The specification of the clone method uses a logical variable S : *Cloneable** to express the freshness of the object which the *clone* method returns. It says in the precondition that every *Cloneable*-object is stored in the sequence S. The postcondition states that the result value does not occur in S, which ensures that it is a new object. The postcondition also states that the new object's x field has received the initial value of the x field of the receiver of the *clone* method.

The interesting part of this example is the creation statement in the *clone* method, which is specified as follows.

```
class Cloneable {
  int x ;

  requires true;
  ensures this.x = old(p);
  C(int p) {
    assert p = old(p);
    this.x := p ;
    assert this.x = old(p);
  }

  requires (∀o : Cloneable • o ∈ S);
  ensures ¬(result ∈ S) ∧ result.x = this.x;
  Cloneable clone() {
    assert (∀o : Cloneable • o ∈ S) ∧ this.x = old(this.x);
    Cloneable u ;
    assert (∀o : Cloneable • o ∈ S) ∧ this.x = old(this.x);
    intermediate ¬(fresh ∈ S) ∧ this.x = old(this.x);
    u := new Clonable(this.x) ;
    assert ¬(u ∈ S) ∧ u.x = old(this.x);
    return u ;
  }
}
```

Figure 6.2: Example: cloneable objects

```
assert (∀o : Cloneable • o ∈ S) ∧ this.x = old(this.x);
intermediate ¬(fresh ∈ S) ∧ this.x = old(this.x);
u := new C(this.x);
assert ¬(u ∈ S) ∧ u.x = old(this.x);
```

Recall that a clause $o \in S$ abbreviates the formula

$$(\exists i \bullet 0 \leq i < \mathsf{length}(S) \wedge o = S[i]) \ .$$

The intermediate keyword reveals that we will employ our second approach to reasoning about object creation. Thus we get two verification conditions for this code fragment.

The first verification condition of this proof outline corresponds to the allocation of the new object, and is defined in (6.18). It is the implication

$$\begin{aligned}(\forall o : \mathit{Cloneable} \bullet o \in S) \wedge \mathsf{this}.x = \mathsf{old}(\mathsf{this}.x) \rightarrow \\ (\mathsf{h}^*(\neg(\mathsf{fresh} \in S) \wedge \mathsf{this}.x = \mathsf{old}(\mathsf{this}.x))[\mathsf{new}(\mathit{Cloneable})/\mathsf{fresh}]) \ . \quad (6.20)\end{aligned}$$

The operation h^* has no effect on its argument because it does not contain any conditional expressions. The following computation reveals the result of the

application of $[\mathsf{new}(\mathit{Cloneable})/\mathsf{fresh}]$ on the clause $\neg(\mathsf{fresh} \in S)$.

$$\begin{array}{l}\neg(\mathsf{fresh} \in S)[\mathsf{new}(\mathit{Cloneable})/\mathsf{fresh}]\\ \equiv \neg(\exists i \bullet 0 \leq i < \mathsf{length}(S) \wedge \mathsf{fresh} = S[i])[\mathsf{new}(\mathit{Cloneable})/\mathsf{fresh}]\\ \equiv \neg(\exists i \bullet 0 \leq i < \mathsf{length}(S) \wedge ((\mathsf{fresh} = S[i])[\mathsf{new}(\mathit{Cloneable})/\mathsf{fresh}]))\\ \equiv \neg(\exists i \bullet 0 \leq i < \mathsf{length}(S) \wedge \mathsf{false})\end{array}$$

The clause $S[i] = \mathsf{fresh}$ is reduced to false because $S[i]$ cannot be equal to the new object fresh. Observe that the resulting formula is equivalent to true. We have

$$\mathsf{this}.x = \mathsf{old}(\mathsf{this}.x)[\mathsf{new}(\mathit{Cloneable})/\mathsf{fresh}]) \equiv \mathsf{this}.x = \mathsf{old}(\mathsf{this}.x)$$

because neither $\mathsf{this}.x$ nor $\mathsf{old}(\mathsf{this}.x)$ involves an expression that may denote the new object. These two results show that (6.20) is a tautology.

The second verification condition of our code fragment corresponds to the invocation of the constructor method. It is therefore based on the specification of the constructor method as given in Fig. 6.2, which says that the receiver's x field gets the initial value of parameter p. The outline of the verification condition can be found in (6.19). The real verification condition is as follows.

$$\begin{array}{r}\mathsf{defined}(\mathsf{this}.x) \wedge \neg(\mathsf{fresh} = \mathsf{null}) \wedge \mathsf{heap}_V \wedge \neg(\mathsf{fresh} \in S) \wedge \lfloor \mathsf{this}.x \rfloor = \mathsf{old}(\mathsf{this}.x)\\ \wedge (\mathsf{true} \rightarrow \mathsf{fresh}.x = \lfloor \mathsf{this}.x \rfloor) \rightarrow\\ \neg(\mathsf{fresh} \in S) \wedge \mathsf{fresh}.x = \mathsf{old}(\mathsf{this}.x) \quad (6.21)\end{array}$$

We did not expand the formulas heap_V and $\lfloor \mathsf{this}.x \rfloor$ for brevity. Note that we have $\lfloor \neg(\mathsf{fresh} \in S) \rfloor \equiv \neg(\mathsf{fresh} \in S)$ because the formula does not quantify over objects, and it also does not contain references to object fields. The list $\bar{z}$ is empty because there are no free occurrences of logical variables in the specification of the constructor method. It is not difficult to see that (6.21) is also a tautology.

6.3 Related Work

The weakest precondition calculus for object allocation in this chapter is an extension of the calculus that was developed by De Boer [dB91, dB99]. Our calculus shows which amendments are necessary in order to handle inheritance and subtype polymorphism.

A sketch of the strongest postcondition for object allocation has also been given by de Boer [dB99]. Ábráham has used his ideas to formulate a cooperation test for object allocation in the context of a multi-threaded subset of Java [Á05, p. 43]. This test involves a global invariant which is absent in our setting. Other differences are caused by the fact that her (local) assertion language does neither support the $\mathsf{old}(.)$ construct nor free occurrences of logical variables. Our research also reveals that the strongest postcondition of object allocation

requires additional type information (cf. the clause Q_2 above) in the context of a language with subtype polymorphism.

The weakest precondition of a statement S with respect to a postcondition Q characterizes the set of states in which a computation of S terminates in a state that satisfies Q. It is impossible to use such a semantical definition of the weakest precondition of object allocation in our logic because we cannot directly express this set in our assertion language. However, if one uses, for example, higher order logic as specification language, then it becomes possible to encode the semantics of object allocation in the assertion language. Thus one can use the given description of the weakest precondition of object allocation for reasoning about object allocation. Von Oheimb [vO01] employs this approach in his Hoare logic for sequential Java. The weakest precondition calculus by Jacobs [Jac04] also works in this way. A disadvantage of this approach is that it blurs the traditional distinction between an axiomatic semantics and an operational semantics. Program verification by means of such a logic requires knowledge of both the Hoare logic and the operational semantics that is used to formalize the behavior of object allocation.

The approach taken by Poetzsch-Heffter and Müller [PHM98] lies somewhere in between our approach and the semantical approach of the previous paragraph. Their specification language supports expressions of the form $new(\$, C)$ and $\$\langle C\rangle$, which denote a fresh object of class C in the present heap $\$$ and the heap that results if $new(\$, C)$ is allocated, respectively. With these expressions they formulate the (possibly weakest) precondition of object allocation. However, an additional non-trivial axiomatization of the properties of these expressions is required to reason about the resulting verification conditions. This suggests that it is more difficult to automatically (dis)prove such formulas.

None of the papers cited in this section (except our own work [PdB05b]) gives a formal account of constructor methods. To the best of our knowledge, we have given the first complete axiomatization of the entire creation process (including both the allocation and the initialization of an object) in this chapter.

Chapter 7

Formal Justification

In the previous chapters we have developed techniques for reasoning about all basic statements in our object-oriented language COORE. In this chapter we bring all these fragments together and paint the full picture: a sound and complete proof outline logic for object-oriented programs.

This chapter consists of three parts. In the first part we present a verification strategy that shows how we can compute the verification conditions of annotated methods. In fact, we will present two strategies: a basic strategy that computes the verification conditions of the annotated statements as defined in Chapter 3, and an optimized strategy that requires less annotation.

The second and third part of this chapter contain a soundness proof and a completeness proof, respectively. The soundness proof is relatively short because many elements of the proof have already been presented in previous chapters. The completeness proof is more interesting and also more elaborate. It is the first completeness proof for an object-oriented proof outline logic with inheritance and subtype polymorphism.

The only other complete program logic for such a language that we know of is the Hoare-like logic for (sequential) Java by Von Oheimb [vO01], which uses the classical Hoare rules for reasoning about method calls. For this reason his logic is not suitable for verification condition generation (cf. Section 5.1.3). Our proof outline logic handles method calls by means of adaptation rules, which are well-suited for verification condition generation. The heart of our completeness proof is a result (Theorem 7.8 below) that shows that these adaptation rules also lead to a complete logic.

7.1 Verification Condition Generation

In Section 3.4 we defined a set of proof obligations for annotated methods (see Table 3.1). We have shown that the validity of this set of proof obligations implies the validity of the interface specification of the method (cf. Theorem

3.12). The set of proof obligations of a method contained Hoare triples as well as assertions. In the preceding three chapters, we have developed techniques that enable us to compute verification conditions for Hoare triples. We will apply these techniques here in this section to define alternative sets of proof obligations (called verification conditions) which will no longer contain Hoare triples. All verification conditions will merely be formulas of our assertion language.

The total set of verification conditions of an annotated program will always be the *union* of the verification conditions of each individual class in the program. Likewise, the set of verification conditions of a particular class is given by the union of the verification conditions of all its methods. The remaining question is therefore how to define the set of verification conditions of annotated methods.

Each definition of the set of verification conditions of annotated methods can be seen as a particular verification strategy. Two strategies are discussed in the remainder of this section. A basic strategy that assumes that each method is fully annotated is defined in Section 7.1.1. Section 7.1.2 contains a more economical strategy that requires fewer annotations.

7.1.1 A Basic Verification Strategy

The following verification strategy takes as input an annotated method as defined in Section 3.3. Additionally, we will assume that each creation statement in the body of the method is preceded by an additional assertion indicated by the intermediate keyword as in the following proof outline schema (cf. Section 6.2).

$$\mathsf{assert}\ Q;\ \mathsf{intermediate}\ Q';\ u := \mathsf{new}\ C(e_1, \ldots, e_n)\ ;\ \mathsf{assert}\ Q'';$$

Recall from Section 3.3 that an annotated method has the following form.

$$\begin{array}{l}\mathsf{requires}\ P;\\ \mathsf{ensures}\ Q;\\ t\ m(\bar{p})\ \{\ \mathsf{assert}\ Q';\ S\ ;\ \mathsf{assert}\ Q'';\ \mathsf{return}\ e)\ \}\end{array}$$

Note that the body S of the method is enclosed by two assertions (its precondition Q' and its postcondition Q''). We call a statement S that is enclosed by two assertions a fully annotated statement. The set of verification conditions of an annotated method includes the verification conditions of its fully annotated body. The main part of the definition of our basic verification strategy consists of the definition of the set of verification conditions of fully annotated statements.

We denote the set of verification conditions of an annotated statement

$$\mathsf{assert}\ Q;\ S;\ \mathsf{assert}\ Q';$$

by $VC(Q, S, Q')$. This function always depends implicitly on the program π in which the statement occurs because, e.g., the verification conditions of method

calls depend on the interface specifications of the corresponding methods, which are defined elsewhere in π.

We start with some simple cases. The verification conditions for assignments are computed using the weakest precondition calculus for assignments in Chapter 4 in the usual way. The verification conditions check whether the preconditions imply the weakest preconditions of the statements with respect to the postconditions.

$$\begin{aligned} VC(Q, t\ u, Q') &= \{Q \rightarrow Q'[\mathsf{def}(t)/u]\} \\ VC(Q, u := e, Q') &= \{Q \rightarrow (\mathsf{defined}(e) \rightarrow Q'[e/u])\} \\ VC(Q, e.x := e', Q') &= \{Q \rightarrow (Q'[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \\ &\qquad \vee \neg\mathsf{defined}(e'))\} \end{aligned}$$

The verification conditions of composed statements are defined in the same way as their sets of proof obligations in Section 3.4.

$$\begin{aligned} VC(Q, S_1; \mathsf{assert}\ Q''; S_2, Q') &= VC(Q, S_1, Q'') \cup VC(Q'', S_2, Q') \\ VC(Q, \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2, Q') &= VC(Q \wedge e, S_1, Q') \\ &\quad \cup VC(Q \wedge e = \mathsf{false}, S_2, Q') \\ VC(Q, \mathsf{while}\ (e)\ \mathsf{assert}\ Q''; \ S, Q') &= VC(Q'' \wedge e, S, Q'') \\ &\quad \cup \{Q \rightarrow Q'', Q'' \wedge e = \mathsf{false} \rightarrow Q'\} \end{aligned}$$

Next, we define the verification conditions of dynamically-bound method calls using the adaptation rule for this type of method call as defined in Section 5.2.3. That is, we will define the set $VC(Q, u := e_0.m(e_1, \ldots, e_n), Q')$. Recall from Section 5.2.3 that $\mathsf{impls}([e_0], m)$ denotes the set of classes that contains implementations of method m to which the call $e_0.m(e_1, \ldots, e_n)$ may be bound. Let $\mathsf{impls}([e_0], m) = \{C_1, \ldots, C_k\}$. Then we have

$$VC(Q, u := e_0.m(e_1, \ldots, e_n), Q') = \{V_1, \ldots, V_k\}\ ,$$

where V_j, for $j \in \{1 \ldots k\}$, is defined as described on page 84. The verification conditions for a call of the form $e_0.m(e_1, \ldots, e_n)$ (i.e., a call to a method which does not return a value) are similar - just drop the substitutions $[\mathsf{result}/u]$ in the verifications conditions.

Finally, we define the verification conditions of creation statements of the form $u := \mathsf{new}\ C(e_1, \ldots, e_n)$. As mentioned above, we assume that each creation statement is preceded by a clause of the form $\mathsf{intermediate}\ Q$, which describes the state after the allocation of the new object. We will use the weakest precondition calculus for object allocation to express the verification conditions for creation statements because this leads to much more concise formulas than the alternative set of verification conditions based on the strongest postconditions of object allocation. The corresponding verification conditions (6.18) and (6.19) are listed in Section 6.2. Hence we have

$$VC(Q, \mathsf{intermediate}\ Q'; u := \mathsf{new}\ C(e_1, \ldots, e_n), Q'') = \{(6.18), (6.19)\}\ .$$

The previous definition finishes our definitions of $VC(Q, S, Q')$. The given definitions together form a nice overview of some of the techniques that we have developed in the previous chapters. Next, we use the set of verification conditions of fully annotated statements to define the verification conditions of annotated methods. The following definition is an obvious variant of Definition 3.5, which defined the set of proof obligations of a method.

Definition 7.1. *The set of verification conditions of an annotated method of the form*

$$\begin{array}{l}\mathsf{requires}\ P;\\ \mathsf{ensures}\ Q;\\ \mathsf{void}\ m(\bar{p})\ \{\ \ \mathsf{assert}\ Q';\ S;\ \mathsf{assert}\ Q'';\ \}\end{array}$$

is the set $\{P \to Q'[./\mathsf{old}(.)], Q'' \to Q\} \cup VC(Q', S, Q'')$.

The set of verification conditions of method that return a value is only slightly more complex.

Definition 7.2. *The set of verification conditions of an annotated method of the form*

$$\begin{array}{l}\mathsf{requires}\ P;\\ \mathsf{ensures}\ Q;\\ t\ m(\bar{p})\ \{\ \ \mathsf{assert}\ Q';\ S;\ \mathsf{assert}\ Q'';\ \mathsf{return}\ e'\ \ \}\end{array}$$

is the set $\{P \to Q'[./\mathsf{old}(.)], Q'' \to (\mathsf{defined}(e) \to Q[e'/\mathsf{result}])\} \cup VC(Q', S, Q'')$.

Our basic verification strategy treats every statement in the same way in the sense that each statement requires a postcondition *and* a precondition. For certain statements we can also compute a valid precondition for every given postcondition. We will use this observation in the following section to define a more economical verification strategy. The soundness of the basic verification strategy is discussed in Section 7.2.

7.1.2 An Advanced Verification Strategy

The basic verification strategy in the previous section takes as input a method implementation in which each statement is fully annotated. This raises the question whether it is possible to define a verification strategy for statements in which some of the annotations are missing. We will go one step further by asking ourselves the question what the minimum set of annotations is for which there exists a verification strategy based on the techniques in this thesis. We will answer that question in this section, and we will also give the corresponding verification strategy.

Ideally, our logic would provide weakest precondition calculi for all statements in the programming language. However, it is customary to reason about while loops using invariants instead of weakest preconditions [Hoa69]. (It is in

principle possible to compute weakest preconditions for loops (cf. [Mey85, pp. 200-211], but this requires a stronger assertion language.) Invariants must be supplied by the reasoner. Alternatively, one could use randomly-generated test runs to try to predict invariants automatically, but that does not always yield a suitable invariant [ECGN01]. Additionally, our logic has no weakest precondition calculus for method calls. Our adaptation rules are based on strongest postconditions. We have explained in Section 5.2.2 that this decision is unavoidable in an object-oriented context.

These two observations reveal which intermediate assertions are indispensable in proof outlines: each loop must be annotated with an invariant, and a precondition must be supplied for every method call (including calls to constructor methods). Furthermore, each method must have an interface specification consisting of a precondition and a postcondition. All other annotations can be computed using weakest precondition calculi. Fig. 7.1 shows a schema of an annotated method with precisely those annotations.

$$
\begin{array}{l}
\mathsf{requires}\ P; \\
\mathsf{ensures}\ Q; \\
t\ m(p_1, \ldots, p_n)\ \{ \\
\quad \ldots \\
\quad \mathsf{assert}\ Q;\ u := e.m(\bar{e}) \\
\quad \ldots \\
\quad \mathsf{while}\ (e)\ \mathsf{assert}\ Q;\ S \\
\quad \ldots \\
\quad \mathsf{intermediate}\ Q;\ u := \mathsf{new}\ C(\bar{e}) \\
\quad \ldots \\
\}
\end{array}
$$

Figure 7.1: A schema of an annotated method indicating the minimum amount of annotation that suffices for our proof outline logic.

We will call a statement that contains precisely the annotations which are indispensable a *sparsely* annotated statement.

Definition 7.3. *The grammar of sparsely annotated statements is as follows.*

$$
\begin{array}{rcl}
S^- \in \mathit{Stat}^- & ::= & t\ u \mid u := e \mid e.x := e \mid S^-\ ;\ S^- \mid \mathsf{assert}\ Q;\ e.m(\bar{e}) \\
& \mid & \mathsf{assert}\ Q;\ u := e.m(\bar{e}) \mid \mathsf{intermediate}\ Q;\ u := \mathsf{new}\ C(\bar{e}) \\
& \mid & \mathsf{if}\ (e)\ S^-\ \mathsf{else}\ S^- \mid \mathsf{while}\ (e)\ \mathsf{assert}\ Q;\ S^-
\end{array}
$$

The verification strategy for sparsely annotated statements does not require that each statement is annotated with both a precondition and a postcondition. For most statements, we can infer a valid precondition for a given postcondition using a weakest precondition calculus. We will call this formula the *inferred* precondition of the statement. In most cases it is the weakest precondition of the statement with respect to the given postcondition. In all other cases we simply

take the additionally supplied assertion. We will denote the inferred precondition of a sparsely annotated statement S^- with respect to a postcondition Q by $\mathsf{ip}(S^-, Q)$. We distinguish the following cases.

$$\begin{aligned}
\mathsf{ip}(t\ u, Q) &= Q[\mathsf{def}(t)/u] \\
\mathsf{ip}(u := e, Q) &= \mathsf{defined}(e) \rightarrow Q[e/u] \\
\mathsf{ip}(e.x := e', Q) &= Q[e'/e.x] \vee \neg\mathsf{defined}(e) \vee e = \mathsf{null} \\
&\qquad \vee \neg\mathsf{defined}(e') \\
\mathsf{ip}(S_1^-\ ;\ S_2^-, Q) &= \mathsf{ip}(S_1^-, \mathsf{ip}(S_2^-, Q)) \\
\mathsf{ip}(\mathsf{assert}\ Q'; e.m(\bar{e}), Q) &= Q' \\
\mathsf{ip}(\mathsf{assert}\ Q'; u := e.m(\bar{e}), Q) &= Q' \\
\mathsf{ip}(\mathsf{intermediate}\ Q'; u := \mathsf{new}\ C(\bar{e}), Q) &= \mathsf{h}^*(Q')[\mathsf{new}(C)/v] \\
\mathsf{ip}(\mathsf{if}\ (e)\ S_1^-\ \mathsf{else}\ S_2^-, Q) &= e \rightarrow \mathsf{ip}(S_1^-, Q) \wedge e = \mathsf{false} \rightarrow \mathsf{ip}(S_2^-, Q) \\
\mathsf{ip}(\mathsf{while}\ (e)\ \mathsf{assert}\ Q'; S^-, Q) &= Q'
\end{aligned}$$

Observe that each creation statement is again preceded by an assertion that describes the intermediate state; The inferred precondition of a creation statement is the weakest precondition of this formula and the allocation of the object that it creates.

The verification strategy for sparsely annotated statements is a function that assigns a set of verification conditions to each pair that consists of a sparsely annotated statement and a postcondition. We denote the set of verification conditions of a sparsely annotated statement S^- with respect to a postcondition Q by $VC^*(S^-, Q)$; the function VC^* is defined by induction on the structure of S^-.

The inferred precondition of a basic assignment S^- with respect to some postcondition Q is its weakest precondition with respect to that postcondition. The weakest precondition is always a valid precondition; we do not need to verification condition to check that. Therefore we have $VC^*(S^-, Q) = \emptyset$ for every basic assignment S^-.

$$\begin{aligned}
VC^*(t\ u, Q) &= \emptyset \\
VC^*(u := e, Q) &= \emptyset \\
VC^*(e.x := e', Q) &= \emptyset
\end{aligned}$$

In composed statements, the inferred preconditions take the roles of the missing assertions.

$$\begin{aligned}
VC^*(S_1^-;\ S_2^-, Q) &= VC^*(S_1^-, \mathsf{ip}(S_2^-, Q)) \cup VC^*(S_2^-, Q) \\
VC^*(\mathsf{if}\ (e)\ S_1^-\ \mathsf{else}\ S_2^-, Q) &= VC^*(S_1^-, Q) \cup VC^*(S_2^-, Q) \\
VC^*(\mathsf{while}\ (e)\ \mathsf{assert}\ Q';\ S^-, Q) &= VC^*(S^-, Q') \cup \{Q' \wedge e \rightarrow \mathsf{ip}(S^-, Q')\} \\
&\qquad \cup \{Q' \wedge e = \mathsf{false} \rightarrow Q\}
\end{aligned}$$

The verification conditions for method calls in our advanced strategy are equal to the corresponding conditions for method calls in the basic strategy. The additional assertion that precedes a method call is the additional parameter

that is required for the basic strategy.

$$VC^*(\mathsf{assert}\ Q'; e_0.m(\bar{e}), Q) = VC(Q', e_0.m(\bar{e}), Q)$$
$$VC^*(\mathsf{assert}\ Q'; u := e_0.m(\bar{e}), Q) = VC(Q', u := e_0.m(\bar{e}), Q)$$

Our basic verification strategy has two verification conditions for creation statements: one that corresponds to the allocation of the new object, and one that corresponds to the constructor method call. We omit the former verification condition in our advanced strategy. This is again possible because the inferred precondition of the creation statement is the weakest precondition of the allocation of the new object with respect to the assertion that describes the state after the allocation of the new object (Q' in the definition below).

$$VC^*(\mathsf{intermediate}\ Q'; u := e_0.m(e_1, \ldots, e_n), Q'') = \{(6.19)\}$$

The annotation in a sparsely annotated statement suffices to verify the method implementation in which the statement occurs. In other words, we are able to verify a method implementation of the form

$$\mathsf{requires}\ P; \mathsf{ensures}\ Q;\ \ t\ m(\bar{p})\ \{\ \ S^-\ \}$$

without additional annotation. The total set of verification conditions of such a method implementation is as follows.

Definition 7.4. *The minimal set of verification conditions of an annotated method of the form*

$$\mathsf{requires}\ P; \mathsf{ensures}\ Q;\ \ \mathsf{void}\ m(\bar{p})\ \{\ \ S^-\ \ \}$$

is $\{P \rightarrow (\mathsf{ip}(S^-, Q)[./\mathsf{old}(.)])\} \cup VC^*(S^-, Q)$.

We can give a similar definition for methods that return a value.

Definition 7.5. *The minimal set of verification conditions of an annotated method of the form*

$$\mathsf{requires}\ P; \mathsf{ensures}\ Q;\ \ t\ m(\bar{p})\ \{\ \ S^-; \mathsf{return}\ e\ \ \}$$

is

$$\{P \rightarrow (\mathsf{ip}(S^-, Q')[./\mathsf{old}(.)])\} \cup VC^*(S^-, Q')\ \ ,$$

where $Q' \equiv \mathsf{defined}(e) \rightarrow Q[e/\mathsf{result}]$.

We hope that the reader recognizes $\mathsf{defined}(e) \rightarrow Q[e/\mathsf{result}]$ as the weakest precondition of the assignment $\mathsf{result} := e$ with respect to the postcondition Q.

7.2 Soundness

The most important part of the formal justification of a program logic is always its soundness proof. Although it may sometimes be economical to work with a logic that is intentionally unsound in some respects [FLL+02], it certainly is folly to work with a logic of which the soundness is ill-documented or even ill-understood. We will prove in this section that the proof outline logic that corresponds to the basic verification strategy as outlined in Section 7.1.1 is sound.

Many parts of our soundness proof have already been described in previous chapters. The main purpose of this section is therefore to show how all these pieces fit together. The overall structure of the proof may not come as a surprise to some because this kind of soundness proof has been given before (see, e.g., [PHM99, vO01]), but we nevertheless feel that the interested reader deserves to see the complete picture.

The most important difference between our soundness proof and existing soundness proofs for Hoare logics is that our soundness proof shows that every method specification in a program holds if all verification conditions of the program are valid, whereas the soundness proof of a Hoare logic usually shows that every derivable Hoare triple regarding the behavior of some statement holds. Another minor difference is that our proof outline logic works with one fixed specification for every method whereas Hoare logics can use different specifications for different calls (cf. rule (5.1) on page 64). The strength of our adaptation rules compensates this loss of flexibility.

The structure of this section is as follows. We will start by stating our main result. In the remainder of the section we explain the structure of its proof and the required intermediate results.

Theorem 7.1. *Let π be a well-formed and well-typed annotated program (cf. Section 2.2.1, 2.2.2 and 3.3) such that all its verification conditions as defined in Section 7.1.1 are valid. Let m be an arbitrary method in some class C in π with precondition P and postcondition Q. Then $\models \{P\}m@C\{Q\}$.*

In other words, we want to prove that the interface specification of every method in π is valid if all its verification conditions hold. For simplicity, we will focus our discussion on methods that return a value. The translation of our definitions and remarks to methods that do not return a value or to constructor methods is straightforward.

Recall from Definition 3.3 that $\models \{P\}m@C\{Q\}$ means that for every initial state (s, h) and every computation $\langle S, (s, h)\rangle \rightarrow (s', h')$ with $\mathcal{E}[\![e]\!](s', h') = v \neq \bot$ we have that $(s, h) \models P$ implies $(s'[\mathsf{result} \mapsto v], h')(s, h) \models Q$, where S is the body of m and e is the expression that denotes its return value. In other words, we must show that every terminating computation of its body that started in a state that satisfies the precondition ends in a state that satisfies the postcondition if we assign the return value to the logical variable result.

The main proof complication is the fact that we are dealing with mutually recursive methods. This means that we cannot prove our claim by means of a standard structural induction on S because the behavior of method calls in S may depend on the correctness of other parts of π. Instead, we will exploit the observation that every *terminating* computation has a *maximum* recursion depth. This enables us to prove our claim by induction on the maximum recursion depth. This approach is also used in several other soundness proofs [PHM99, vO01].

For this proof technique we need a variant of the operational semantics in Section 2.2.3 that records the maximum recursion depth in a computation. This new operational semantics has formulas of the form

$$\langle S, (s,h)\rangle \overset{r}{\dashrightarrow} (s',h') \ ,$$

which say that a computation of statement S that starts in the state (s,h) terminates in the state (s',h') and that the maximum recursion depth throughout the computation is r.

Recording the recursion depth in an operational semantics is quite simple. An interesting example is the new rule for methods calls. In this rule, we increment the recursion depth parameter by one because a method call moves the computation to the next recursion level.

$$\frac{\begin{array}{c}\mathcal{E}[\![e_0]\!](s,h) = o = (C,i) \\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for } i \in \{1 \ldots n\} \\ \mathcal{E}[\![e]\!](s',h') = v \neq \bot \\ \mathsf{meth}(C,m) \equiv t\ m(p_1,\ldots,p_n)\{\ S\ \mathsf{return}\ e\ \} \\ \langle S, (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h)\rangle \overset{r}{\dashrightarrow} (s',h')\end{array}}{\langle u := e_0.m(e_1,\ldots,e_n), (s,h)\rangle \overset{r+1}{\dashrightarrow} (s[u \mapsto \mathcal{E}[\![e]\!](s',h')], h')} \qquad (MC')$$

The other rules of the new semantics are displayed in Figure 7.2. The additional recursion depth parameter does not influence the existence of a computation. Therefore we can prove the following lemma.

Lemma 7.2. *For every statement S, and every initial state (s,h) and final state (s',h'), we have*

$$\langle S, (s,h)\rangle \to (s',h') \iff \exists r \in \mathbb{N}. \langle S, (s,h)\rangle \overset{r}{\dashrightarrow} (s',h') \ .$$

Proof. Both implications can be proved by induction on the lengths of the derivations of their antecedents. □

Our next step is to define a notion of correctness of method specifications that restricts the validity of a method specification to computations of its body which do not exceed a particular recursion depth.

Definition 7.6. *Let S be the body of method m in class C. Let e be the expression that denotes its return value. Then we say that the Hoare triple*

$$\frac{}{\langle t\ u, (s,h)\rangle \overset{0}{\dashrightarrow} (s[u \mapsto \mathsf{init}([u])], h)} \qquad (VA')$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = v \neq \bot}{\langle u := e, (s,h)\rangle \overset{0}{\dashrightarrow} (s[u \mapsto v], h)} \qquad (LA')$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = o \notin \{\bot, null\} \quad \mathcal{E}[\![e']\!](s,h) = v \neq \bot \quad \mathsf{origin}([e], x) = D}{\langle e.x := e', (s,h)\rangle \overset{0}{\dashrightarrow} (s, h[o.x_D \mapsto v])} \qquad (FA')$$

$$\frac{\langle S_1, (s,h)\rangle \overset{r}{\dashrightarrow} (s'', h'') \quad \langle S_2, (s'', h'')\rangle \overset{r'}{\dashrightarrow} (s', h')}{\langle S_1;\ S_2, (s,h)\rangle \overset{\mathsf{max}(r,r')}{\dashrightarrow} (s', h')} \qquad (SC')$$

$$\frac{\begin{array}{c} o = (C, i) \quad h(o) \text{ is undefined} \\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for } i \in \{1 \ldots n\} \\ \mathsf{constr}(C) \equiv C(p_1, \ldots, p_n)\ \{\ S\ \} \\ \langle S, (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h \cdot [o \mapsto init(C)])\rangle \overset{r}{\dashrightarrow} (s', h') \end{array}}{\langle u := \mathsf{new}\ C(e_1, \ldots, e_n), (s,h)\rangle \overset{r+1}{\dashrightarrow} (s[u \mapsto o], h')} \qquad (OC')$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = tt \quad \langle S_1, (s,h)\rangle \overset{r}{\dashrightarrow} (s', h')}{\langle \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2, (s,h)\rangle \overset{r}{\dashrightarrow} (s', h')} \qquad (IF'_1)$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = ff \quad \langle S_2, (s,h)\rangle \overset{r}{\dashrightarrow} (s', h')}{\langle \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2, (s,h)\rangle \overset{r}{\dashrightarrow} (s', h')} \qquad (IF'_2)$$

$$\frac{\mathcal{E}[\![e]\!](s,h) = ff}{\langle \mathsf{while}\ (e)\ S, (s,h)\rangle \overset{0}{\dashrightarrow} (s,h)} \qquad (WH'_1)$$

$$\frac{\begin{array}{c} \mathcal{E}[\![e]\!](s,h) = tt \\ \langle S, (s,h)\rangle \overset{r}{\dashrightarrow} (s'', h'') \\ \langle \mathsf{while}\ (e)\ S, (s'', h'')\rangle \overset{r'}{\dashrightarrow} (s', h') \end{array}}{\langle \mathsf{while}\ (e)\ S, (s,h)\rangle \overset{\mathsf{max}(r,r')}{\dashrightarrow} (s', h')} \qquad (WH'_2)$$

Figure 7.2: An operational semantics that records the maximum recursion depth.

$\{P\}m@C\{Q\}$ is r-valid, which is denoted by $\models_r \{P\}m@C\{Q\}$, if and only if for every state (s,h) and every computation $\langle S,(s,h)\rangle \xrightarrow{r'} (s',h')$ such that $r' \leq r$, $(s,h) \models P$ and $\mathcal{E}[\![e]\!](s',h') = v \neq \bot$ we have that $(s'[\mathsf{result} \mapsto v],h')(s,h) \models Q$.

We can define a similar correctness notion for ordinary Hoare triples.

Definition 7.7. *A Hoare triple $\{Q\}S\{Q'\}$ is r-valid, denoted by $\models_r \{Q\}S\{Q'\}$, if and only if for every state (s,h), every compatible freeze state (s',h'), and every computation $\langle S,(s,h)\rangle \xrightarrow{r'} (s'',h'')$ with $r' \leq r$ we have $(s,h)(s',h') \models Q$ implies $(s'',h'')(s',h') \models Q'$.*

The relation between $r-$*validity* and total validity of specifications is straightforward.

Lemma 7.3. *For every method specification $\{P\}m@C\{Q\}$ we have that*

$$\models \{P\}m@C\{Q\} \iff \forall r \in \mathbb{N}.\ \models_r \{P\}m@C\{Q\}\ .$$

Proof. Trivial using Lemma 7.2. □

The following lemma is also obvious.

Lemma 7.4. *For every method specification $\{P\}m@C\{Q\}$ and every pair r, r' of recursion depths we have*

$$r < r' \implies (\models_{r'} \{P\}m@C\{Q\} \implies \models_r \{P\}m@C\{Q\})\ .$$

The main lemma that we will prove in this section says that we can prove the $r-$*validity* of every method specification in a program for every recursion depth r. To prove this lemma we first need another lemma that states that we can prove the r-validity of the specification of every annotation method body if we know that every method specification in the program is at least $r-1$-valid.

Lemma 7.5. *Let π be a well-formed and well-typed annotated program as in Theorem 7.1 such that all its verification conditions as defined in Section 7.1.1 are valid. Let $\{P_1\}m_1@C_1\{Q_1\}$, ..., $\{P_k\}m_k@C_k\{Q_k\}$ be all specifications of methods in π. Then we have for every $r \in \mathbb{N}$ and every annotated statement* assert Q'; S; assert Q'; *in π that if*

$$r > 0 \implies \models_{r-1} \{P_1\}m_1@C_1\{Q_1\}, \ldots, \models_{r-1} \{P_k\}m_k@C_k\{Q_k\}$$

then $\models_r \{Q'\}S\{Q''\}$.

Proof. By natural induction on the recursion depth r followed by structural induction on S. We must prove that the validity of the verification conditions of π ensures that for every computation $\langle S,(s,h)\rangle \xrightarrow{r} (s',h')$ we have that $(s,h)(s'',h'') \models Q'$ implies $(s',h')(s'',h'') \models Q''$ for every compatible freeze state (s'',h'').

We first discuss the base case where $r = 0$. If S is a simple assignment we can either refer to Theorem 4.5, Theorem 4.7 or Theorem 4.12. Lemma 7.2 can be used in each case to show that a computation of S with recursion depth 0 corresponds to a 'real' computation of S with the same initial and final state. If S is a sequential statement $S_1; S_2$ we can easily prove our claim using the induction hypothesis (cf. Lemma 3.9).

Note that S cannot be a creation statement or a method call because such statements have a recursion depth greater than 0. If S is a conditional statement or a while-loop we must again use the induction hypothesis. These cases are similar to the proofs of Lemma 3.10 and Lemma 3.11.

For the induction step we assume that our claim holds for recursion depth r and we show that it also holds for recursion depth $r+1$. More in particular, we must show that for every fully annotated statement $\mathsf{assert}\ Q';\ S\ \mathsf{assert}\ Q''$; we have $\models_{r+1} \{Q'\}\ S\ \{Q''\}$. We prove this claim again using structural induction on S. Note that S cannot be an assignment because $r+1$ cannot match with the recursion depth 0 of an ordinary assignment.

Clearly, the most interesting case is the one where $S \equiv u := e_0.m(e_1, \ldots, e_n)$. We must prove that $\models_{r+1} \{Q'\}u := e_0.m(e_1, \ldots, e_n)\{Q''\}$. For this purpose, we consider an arbitrary computation

$$\langle u := e_0.m(e_1, \ldots, e_n), (s, h)\rangle \overset{r'}{\dashrightarrow} (s', h')$$

such that $0 < r' \leq r+1$ and $(s, h)(s'', h'') \models Q'$ for some freeze state (s'', h''). Our goal now is to show that $(s', h')(s'', h'') \models Q''$.

For every method implementation $m@D$ that may be bound to this call we have that $\models_r \{P'\}m@D\{Q'\}$, where P' and Q' are the fixed precondition and postcondition of $m@D$ in π, follows from the induction hypothesis. By Lemma 7.4 this implies that $\models_{r'-1} \{P'\}m@D\{Q'\}$. We can then finish the proof of this case along the lines of the soundness proof of the adaptation rule (Theorem 5.11). Note that this theorem is slightly weaker than what is required for this case because it assumes that we have $\models \{P'\}m@D\{Q'\}$ for every method that may be bound to the call instead of the weaker assumption $\models_{r'-1} \{P'\}m@D\{Q'\}$ that we have here. However, an inspection of the proof reveals that the weaker assumption also suffices to complete the proof.

If S is a creation statement we can prove our claim along the lines of the proof of Theorem 6.19. The proofs of the cases where S is a control statement are only slightly more complex for the induction step than the corresponding proofs for the base case. □

We now come to the above-mentioned main lemma that states that every method specification is $r - \mathit{valid}$ for every recursion depth r if the verification conditions hold.

Lemma 7.6. *Let π be a well-formed and well-typed annotated program as in Theorem 7.1 such that all its verification conditions as defined in Section 7.1.1*

are valid. Let $\{P_1\}m_1@C_1\{Q_1\}$, ..., $\{P_k\}m_k@C_k\{Q_k\}$ *be all method specifications of methods in* π. *Then we have for every* $r \in \mathbb{N}$ *that*

$$\models_r \{P_1\}m_1@C_1\{Q_1\}, \ldots, \models_r \{P_k\}m_k@C_k\{Q_k\} \ .$$

Proof. By natural induction on the recursion depth r. Let $\{P\}\ m@C\ \{Q\}$ be an arbitrary method specification from the given set of method specifications. Let S be its body, and let e be the expression that denotes its return value. We assume that the body of method m is as follows.

$$\mathsf{assert}\ Q';\ S;\ \mathsf{assert}\ Q'';\ \mathsf{return}\ e$$

The proof pattern of the base case and the induction step is the same. We must in each step prove that for some recursion depth r we have that for every state (s,h) and every computation $\langle S,(s,h)\rangle \overset{r}{\dashrightarrow} (s',h')$ such that $(s,h) \models P$ and $\mathcal{E}[\![e]\!](s',h') = v \neq \bot$ we have $(s'[\mathsf{result} \mapsto v],h')(s,h) \models Q$. From the first verification condition of the method body (see Def. 7.2) we get that $(s,h) \models P$ implies $(s,h)(s,h) \models Q'$. Next, we use Lemma 7.5 to prove that $(s',h')(s,h) \models Q''$. Finally, we use the second verification condition to show that $(s',h')(s,h) \models Q$. In this step we use the observation that $\mathcal{E}[\![e]\!](s',h') \neq \bot$ implies $(s',h')(s,h) \not\models \neg\mathsf{defined}(e)$. □

Our main theorem 7.1 follows directly from Lemma 7.3 and Lemma 7.6.

7.3 Relative Completeness

Investigating the relative completeness of a Hoare logic is the classical way to evaluate its strength [Apt81]. Relative completeness is a formal property of a Hoare logic that ensures that any failed attempt to verify the correctness of a program is not caused by a weakness of one of its rules or axioms.

Completeness of a logic means in general that every valid formula can be derived within the logic. Thus completeness of a Hoare logic boils down to the property that every valid Hoare triple $\{P\}\ S\ \{Q\}$ can be derived using the rules and axioms of the logic.

It is customary to prove completeness for Hoare logics under two additional assumptions. The first assumption is that every valid formula of the assertion language is an axiom of the Hoare logic [Apt81]. This assumption enables the completeness proof to focus on the strength of the Hoare rules instead of the underlying proof system for the assertion language. The second assumption concerns the expressiveness of the assertion language; one only proves completeness for interpretations of the assertion language that allow one to express the strongest postcondition or the weakest precondition of every statement in the assertion language [Coo78]. This completeness notion is known as *relative* completeness.

Our logic is based on a fixed interpretation of the assertion language. We can therefore eliminate the second assumption by showing that either the strongest

postcondition or the weakest precondition can be expressed in our assertion language. We will further discuss this issue in Section 7.3.4.

Before we plunge into the details of the completeness proof of our proof outline logic we first sketch the outline of Gorelick's seminal completeness proof for recursive programs [Gor75]. We will use his proof as a blueprint for the construction of our completeness proof, developing many elements of Gorelick's proof in order to make them fit for the object-oriented world.

7.3.1 Gorelick's Completeness Proof for Recursive Programs

To prove that a particular Hoare logic is complete we must consider an arbitrary valid Hoare triple $\{P\}\ S\ \{Q\}$ and show that we can derive it. The complexity of this task largely depends, of course, on which statements S the language supports, and on which preconditions P and postconditions Q can be expressed in the assertion language. For simple languages without recursive procedures (or methods) one can usually prove completeness by means of a straightforward induction on the complexity of S. A similar proof can also be given for languages with non-recursive procedures by first replacing every procedure call in S by the corresponding procedure body.

The situation changes, however, when the programming language supports recursive procedures. For to derive a particular Hoare triple regarding a call to a recursive procedure one must first pick a particular procedure specification and then show that the chosen specification suffices to prove that the procedure body indeed satisfies this specification (cf. rule 5.1 on page 64). And the chosen specification must of course also be strong enough to derive the initial Hoare triple of the procedure from it. In a completeness proof one must stipulate a suitable procedure specification for every valid Hoare triple of a procedure call.

The main contribution of Gorelick's seminal completeness proof for recursive procedures [Gor75] is its introduction of a kind of procedure specification that suffices to derive every valid Hoare triple. The specifications that Gorelick used are known as Most General Formulas (MGFs). But before we explain what an MGF is, we must first describe its context because an MGF is always tailored to a particular programming language.

We will assume in this section that we are dealing with a simple sequential programming language with (mutually) recursive parameterless procedures (as in Section 5.1.1). Let

$$p_1 \Leftarrow S_1, \ldots, p_n \Leftarrow S_k$$

be a set of k mutually recursive procedure declarations, where $p \Leftarrow S$ indicates that statement S is the body of procedure p. We assume here that every statement S_i with $i \in \{1, \ldots, k\}$ only modifies elements of the set of global variables $\{x_1, \ldots, x_n\}$.

The key element of an MGF is a formula that 'freezes' the initial state by claiming a correspondence between the program variables and a set of 'freeze'

variables. The freeze formula of our simple language has the form $\bigwedge_{i=1}^{n}(x_i = z_i)$, where z_i is a logical variable of the same type as x_i, for $i \in \{1 \ldots n\}$. The Most General (correctness) Formula of a statement S is then the formula

$$\{\bigwedge\nolimits_{i=1}^{n}(x_i = z_i)\}S\{\mathsf{sp}(S, \bigwedge\nolimits_{i=1}^{n}(x_i = z_i))\} \ .$$

The postcondition $\mathsf{sp}(S, \bigwedge_{i=1}^{n}(x_i = z_i))$ denotes the strongest postcondition of S with respect to the precondition $\bigwedge_{i=1}^{n}(x_i = z_i)$.

The rest of Gorelick's proof can be divided in two parts. First, he proves that every valid correctness formula $\{P\}S\{Q\}$ can be derived using these Most General Formulas by induction on the complexity of S. Naturally, the most interesting case involves a procedure call because there one must bring the suggested procedure specifications into play. And secondly, he shows that it is possible to derive the MGF for every procedure in the program using the recursion rule. This latter step is able to exploit the result of the first part because an MGF is by definition a valid correctness formula.

7.3.2 The Structure of Our Completeness Proof

The structure of the completeness proof of our proof outline logic differs from that of Gorelick's proof in certain respects. These dissimilarities are due to the difference in scope between proof outline logics and Hoare logics. Hoare logics can be used to derive valid specifications of arbitrary statements, whereas a proof outline always corresponds to a complete method implementation.

What we will do is the following. First, we will demonstrate that these Most General Formulas enable all clients of a method to derive every valid specification of a arbitrary invocation of the method. And secondly, we will prove that we can annotate method bodies in such a way that we get a valid proof outline for every method that is annotated with its Most General Formula.

Our first task will be to find an object-oriented counterpart of the freeze formula in Gorelick's proof. Gorelick's freeze formula only suffices for programs with a finite set of global variables. In order to freeze the initial state of a method execution we will have to find a way to store the internal states of all its objects (a set of arbitrary size) in a finite set of logical variables.

7.3.3 Freeze Formulas

The freeze formula $\bigwedge_{i=1}^{n}(x_i = z_i)$ in Gorelick's proof freezes the initial values of all the variables that may be changed during program execution by 'storing' them in freeze variables. Thus it ensures that these values are still available in the final state of a computation.

The main complication for an object-oriented freeze formula are the dynamic boundaries of states of object-oriented programs. The internal states of all objects must also be stored in freeze variables, and the number of objects in a state is not fixed. Thus the size of states is not fixed. For this reason

object-oriented languages belong to the category of languages with dynamically allocated variables. The techniques that we describe in this section can also be applied in completeness proofs for program logics of other languages in this category [dBP04].

The fact that we do not know *a priori* how many objects exist in a particular heap does not mean that it is impossible to freeze an object-oriented state. The key to an object-oriented freeze formula is the observation that the set of objects that exists in a particular state is *finite*. Hence we can use a variable of type object^* (a finite sequence of objects) to store all the objects in a particular state. Using this observation we can introduce freeze variables in the same way as we used logical variables in Section 5.2.2 to model the dual heap.

Let $\mathcal{F}$ be a fresh logical variable of type object^*. This variable will freeze all objects that exist in the initial state. Let $\mathcal{F}(x_C)$ be a fresh logical variable of type $[x]^*$ for every field x defined in some class C. We will use the i-th position of the sequence denoted by $\mathcal{F}(x_C)$ to store the value of the x field of the object at position i in $\mathcal{F}$ (if this object inherits field x).

Next, we must express the correspondence between the program variables and the freeze variables in a formula. The first clause of our freeze formula states that every object is stored in the sequence $\mathcal{F}$.

$$(\forall z : \mathsf{object} \bullet z = \mathsf{null} \vee z \in \mathcal{F}) \tag{7.1}$$

Our second clause states, for every field x in some class C, where the values of the x- fields of all C-objects can be found in $\mathcal{F}(x_C)$.

$$\bigwedge_{x_C} (\forall i \bullet 0 \leq i < \mathsf{length}(\mathcal{F}) \wedge \mathcal{F}[i] \ \mathsf{instanceof}\ C \rightarrow ((C)\mathcal{F}[i]).x = \mathcal{F}(x_C)[i]) \tag{7.2}$$

The cast in $((C)\mathcal{F}[i]).x$ is necessary to ensure that the expression denotes the value of the x field declared in class C; recall that COORE supports field shadowing, which means that superclasses of C may also contain field declarations with the same identifier.

We will denote the conjunction of (7.1) and (7.2) by *init*.

Interestingly, we do *not* have to freeze the values of the formal parameters and the implicit this-parameter. The reason for this is that we can already refer to the initial values of these variables in the final state by means of expressions of the form $\mathsf{old}(p)$ and $\mathsf{old}(\mathsf{this})$. This raises the question whether the freeze formula is perhaps redundant since we can also use expressions of the form $\mathsf{old}(e.x)$ to denote the initial value of field x of object e. The answer to this question is 'no', and the reason is as follows. We must at least be able to refer to the initial values of the fields of all reachable objects (unreachable objects cannot be modified by a method). However, we do not know *a priori* how many objects are reachable in the initial state. Take, for example, the set of objects in a linked list and assume that we want to freeze the value of the *next* link of each node in the list. It is not clear how many expressions of the form $\mathsf{old}(e.next)$ we need, for that depends on the length of the list in a particular state. So the freeze formula remains indispensable.

It is also worthwhile to observe that there is an important difference between the role of the freeze variables $\mathcal{F}$ and $\mathcal{F}(x_C)$ on the one hand, and the dual heap variables $\mathcal{H}$ and $\mathcal{H}(x_C)$ (see Section 5.2.2) on the other hand. We use the former variables in the precondition of a method to freeze the initial state. The latter variables are employed in the verification conditions of method calls. It is technically impossible to identify both sets of variables since the adaptation rule requires that the dual heap variables do not occur in the specification of the corresponding method.

7.3.4 Strongest Postconditions

Recall from Section 7.3.1 that Gorelicks's Most General Formula has the form $\{F\}S\{\mathsf{sp}(S,F)\}$ where $\mathsf{sp}(S,F)$ denotes the strongest postcondition of S and the freeze formula F. Our counterpart of his MGF will also be based on the strongest postcondition of a statement. Moreover, we will need assertions that express strongest postconditions of statements with respect to arbitrary preconditions to be able to build valid proof outlines for methods in our completeness proof. For these reasons we will study the strongest postconditions of object-oriented statements in this section.

We will use the following 'semantical' definition of a strongest postcondition in our completeness proof.

Definition 7.8. *The strongest postcondition of a statement S and a precondition Q, denoted by $\mathsf{sp}(Q,S)$, is the set of pairs of states (s,h) and compatible freeze states (s',h') defined by*

$$\{\langle(s,h),(s',h')\rangle \mid \exists(s_0,h_0) : \langle S,(s_0,h_0)\rangle \rightarrow (s,h) \text{ and } (s_0,h_0)(s',h') \models Q)\} \ .$$

In other words, the strongest postcondition of an assertion Q with respect to a statement S is the set of pairs $\langle(s,h),(s',h')\rangle$, consisting of a state (s,h) and a compatible freeze state (s',h'), for which there exists an initial state (s_0,h_0) of a computation of S that terminates in (s,h) in which the precondition Q holds.

An important question concerning our strongest postcondition notion is whether there is, for every statement S and precondition Q, an assertion Q' that expresses it. That is, we would like to know whether there is an assertion Q' such that for every state (s,h) and freeze state (s',h'), we have

$$(s,h)(s',h') \models Q' \iff \langle(s,h)(s',h')\rangle \in \mathsf{sp}(Q,S) \ .$$

This issue has been studied rigorously by Tucker and Zucker for programming languages with Abstract Data Types [TZ88]. They showed that the strongest postcondition can be expressed in the weak second-order language that is obtained by extending a first-order language with finite sequences. De Boer has proved comparable results for a basic object-oriented language [dB91]. Our assertion language is similar to the languages considered in these two studies. For this reason we will not explore this question further, and merely assume that

the strongest postcondition can be expressed in our assertion language. Moreover, we will silently identify the strongest postcondition with the assertion that expresses it.

The definition of the strongest postcondition in this section differs from the definition that we used in Section 4.3 and in Section 6.1.2, where we simply defined the strongest postcondition of a statement S and a precondition Q as the strongest assertion Q' that is still a valid postcondition of S with respect to Q. The above-mentioned results can be used to show, however, that the two definitions are equivalent.

We have the following standard result.

Theorem 7.7. *For every statement S and precondition Q we have*

$$\models \{Q\}S\{\mathsf{sp}(Q,S)\} \ .$$

Proof. A direct consequence of Def. 3.2 and Def. 7.8. □

7.3.5 Most General Method Specifications

With the freeze formulas and the strongest postcondition notion from the previous two sections we can build a kind of method specification that will enable us to derive every valid specification regarding method calls. We will call such a method specification a Most General Method Specification (MGMS). It is a pair $\langle P, Q\rangle$ consisting of a precondition P and a postcondition Q.

Definition 7.9. *The Most General Method Specification (MGMS) of a method is*

$$\begin{array}{rcl}
\mathrm{MGMS}(\mathsf{void}\ m(\bar{p})\{\ S\ \}) & = & \langle \mathit{init}, (\exists \bar{z}' \bullet \mathsf{sp}(\mathit{init}^+(\bar{p}), S)[\bar{z}'/\bar{u}'])\rangle \\
\mathrm{MGMS}(C(\bar{p})\{\ S\ \}) & = & \langle \mathit{init}, (\exists \bar{z}' \bullet \mathsf{sp}(\mathit{init}^+(\bar{p}), S)[\bar{z}'/\bar{u}'])\rangle \\
\mathrm{MGMS}(t\ m(\bar{p})\{\ S\ \mathsf{return}\ e\ \}) & = & \langle \mathit{init}, (\exists \bar{z}' \bullet ((\mathsf{sp}(\mathit{init}^+(\bar{p}), S) \wedge \mathsf{defined}(e) \\
 & & \qquad \wedge\ \mathsf{result} = e)[\bar{z}'/\bar{u}'])\rangle
\end{array}$$

where $\mathit{init}^+(\bar{p})$, for $\bar{p} = p_1, \ldots, p_n$, denotes the formula

$$\mathit{init} \wedge \bigwedge_{i=1}^{n} (p_i = \mathsf{old}(p_i)) \ .$$

(For init, see Sect. 7.3.3).

Observe that each postcondition has the form $(\exists \bar{z}' \bullet Q[\bar{z}'/\bar{u}'])$. This form ensures that the postcondition has no occurrences of local variables outside expressions of the form $\mathsf{old}(e)$, which is an important condition for postconditions in our logic. We implicitly assume each time that $\bar{u}'$ is a sequence of the local variables in Q (outside expressions of the form $\mathsf{old}(e)$), and that $\bar{z}'$ is a corresponding sequence of *fresh* logical variables. The substitution $[\bar{z}'/\bar{u}']$ replaces every local variable in Q by its corresponding logical variable in $\bar{z}'$. This does not result in any loss of meaning because the final values of local variables are

always discarded when a method returns. Hence we may as well replace them by arbitrary fresh placeholders. The freshness of the logical variables in $\bar{z}'$ also ensures that the existential quantifiers around $Q[\bar{z}'/\bar{u}']$ do not alter the meaning of the corresponding method specification.

We use expressions of the form $\mathsf{old}(p)$ to freeze the initial values of the formula parameters. The MGMS of a method that returns a value has two additional clauses, which describe its result value.

Justification

In order to justify our definition of the Most General Method Specification of a method we will show that these specifications enables us to prove every valid specification regarding a call. We will only examine the most interesting case, which concerns a Hoare triple of the form $\{Q\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q'\}$.

The following theorem states that every verification condition of the adaptation rule for dynamically-bound method calls holds if we try to prove a valid Hoare triple concerning a call $u := e_0.m(e_1, \ldots, e_n)$, and if the corresponding method implementations are all annotated with their Most General Method Spefications. Thus it justifies our MGMS definition.

Theorem 7.8. *Let $\{Q\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q'\}$ be a valid Hoare triple, and let $\mathsf{impls}([e_0], m) = \{C_1, \ldots, C_k\}$. We assume that the implementation of method m in class C_j has the form $t_j\ m(\bar{p}_j)\{\ S_j\ \mathsf{return}\ e_j\ \}$, for every $j \in \{1 \ldots k\}$. Then we have, also for every $j \in \{1 \ldots k\}$, that $\models (VC_j)$, where P_j and Q_j are the precondition and postcondition of the MGMS of the implementation of method m in class C_j. (For (VC_j), see page 84.)*

Proof. We must prove that every verification condition (VC_j) holds. We will consider an arbitrary verification condition (VC_j) in our proof. For brevity, we will drop all the subscript j's in this particular (VC_j). Let

$$Q'' \equiv (\exists \bar{z}' \bullet (\mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}'])\ .$$

Then we must show that

$$\lfloor \mathsf{boundto}(e_0, C, m) \rfloor \wedge \lfloor (\bigwedge\nolimits_{i=0}^{n} \mathsf{defined}(e_i)) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V \wedge \lfloor Q \rfloor \quad (7.3)$$

and the implication

$$(\forall \bar{z} \in \mathcal{H} \bullet \mathsf{g}(init \wedge \mathsf{rec} = \mathsf{this}) \rightarrow Q''[\mathsf{rec}/\mathsf{this}][\mathsf{g}(.)/\mathsf{old}(.)])) \quad (7.4)$$

together imply $Q'[\mathsf{result}/u]$.

The outline of the proof is as follows. We will consider an arbitrary state (s, h) and a compatible freeze state (s', h') that satisfy both (7.3) and (7.4). Then we perform the following steps.

1. To be able to exploit (7.4) we choose proper values for the logical variables in $\bar{z}$. The required result is a new stack s^* that satisfies $(s^*, h) \models \bar{z} \in \mathcal{H}$.

2. Secondly, we prove that the antecedent $\mathsf{g}(\mathit{init} \wedge \mathsf{rec} = \mathsf{this})$ of the implication in (7.4) holds in this state.
3. The resulting state then also satisfies the consequent of the implication in (7.4), which states that there exists a computation of the body S from some initial state to our modified state.
4. Next, we show that we can extend this computation of the body to a computation of the call.
5. We will prove that the initial state of this computation satisfies Q.
6. Hence Q' holds in the final state of the computation.
7. Our final step shows that this implies that $Q'[\mathsf{result}/u]$ holds in the start state of our proof.

In our first step we assign values to the logical variables in $\bar{z}$ to obtain a state that satisfies the antecedent $\mathsf{g}(\mathit{init} \wedge \mathsf{rec} = \mathsf{this})$ of (7.4). Equation (7.4) also requires that variable $z \in \bar{z}$ with an object type has a value that is part of the dual heap $\mathcal{H}$. By an object type we mean either a class name C or a sequence type C^*. We must prove that $z \in \mathcal{H}$ or $z \sqsubseteq \mathcal{H}$ for every logical variable z with an object type.

Recall that $\bar{z}$ contains the logical variables that occur free in *init* and in Q''. The logical variables in *init* are $\mathcal{F}$, $\mathcal{F}(x_C)$ (for every field x in some class C), and rec. The postcondition Q'' may potentially also contain other logical variables.

The precondition *init* assumes that $\mathcal{F}$ contains every object in the initial heap, and that every variable $\mathcal{F}(x_C)$ contains the corresponding values of field x. The verification condition assumes that these values are stored in the variables $\mathcal{H}$ and $\mathcal{H}(x_C)$, respectively. Therefore we build our new stack s^* from s by assigning $s(\mathcal{H})$ to $\mathcal{F}$, and $s(\mathcal{H}(x_C))$ to $\mathcal{F}(x_C)$ for every field x in some class C. Moreover, s^* assigns the value of $\mathcal{N}[\![\lfloor e_0 \rfloor]\!](s,h)$ to rec.

Observe that $s^*(\mathcal{F}) = s(\mathcal{H})$ ensures that $(s^*,h) \models \mathcal{F} \sqsubseteq \mathcal{H}$. From the clause heap_V in $(s,h) \models (7.3)$ follows that $(s^*,h) \models \mathcal{F}(x_C) \sqsubseteq \mathcal{H}$ for every logical variable $\mathcal{F}(x_C)$ that references a sequence of objects (cf. Eq. (5.20)). Finally, $(s^*,h) \models \mathsf{rec} \in \mathcal{H}$ follows from $(s,h) \models (7.3)$ by Lemma 5.1.

For the remaining logical variables in $\bar{z}$ we make a case distinction. For every logical variable $z \in \bar{z}$ which occurs in either Q or Q' we simply choose $s^*(z) = s(z)$. Thus we ensure that we also have $(s^*,h)(s',h') \models (7.3)$. This value of z is possible for if z has an object type then we already have $(s,h) \models z \in \mathcal{H}$ or $(s,h) \models z \sqsubseteq \mathcal{H}$ because in that case $z \in V$. For every other logical variable $z \in \bar{z}$ we assume that $s^*(z)$ is the default value of z's type. Thus $s^*(z)$ is *null* or the empty sequence if z has an object type.

In the second step of our proof we show that $(s^*,h) \models \mathsf{g}(\mathit{init} \wedge \mathsf{rec} = \mathsf{this})$. We have

$$
\begin{array}{ll}
\mathsf{g}(\mathit{init} \wedge \mathsf{rec} = \mathsf{this}) & \{\ \text{def. g}\ \} \\
= \lfloor (\mathit{init} \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}] \rfloor & \{\ \text{def. } [e_0, \bar{e}/\mathsf{this}, \bar{p}],\ \lfloor . \rfloor\ \} \\
= \lfloor (\mathit{init}[e_0, \bar{e}/\mathsf{this}, \bar{p}]) \rfloor \wedge \mathsf{rec} = \lfloor e_0 \rfloor\ . &
\end{array}
$$

It is clear that $(s^*,h) \models \mathsf{rec} = \lfloor e_0 \rfloor$ holds due to the value of $s^*(\mathsf{rec})$. Recall

that *init* is the conjunction of (7.1) and (7.2). The operation $[e_0, \bar{e}/\mathsf{this}, \bar{p}]$ has no effect on both formulas because they contain no occurrences of this or $\bar{p}$. For (7.1) we get

$$\lfloor(\forall z : \mathsf{object} \bullet z = \mathsf{null} \vee z \in \mathcal{F})\rfloor \\ = (\forall z : \mathsf{object} \bullet (z = \mathsf{null} \vee z \in \mathcal{H}) \rightarrow (z = \mathsf{null} \vee z \in \mathcal{F})) \ .$$

The latter formula clearly holds because $s^*(\mathcal{F}) = s^*(\mathcal{H})$. For (7.2) we get

$$\lfloor(\forall i \bullet 0 \leq i < \mathsf{length}(\mathcal{F}) \wedge \mathcal{F}[i] \ \mathsf{instanceof} \ C \rightarrow ((C)\mathcal{F}[i]).x = \mathcal{F}(x_C)[i])\rfloor \\ = \\ (\forall i \bullet 0 \leq i < \mathsf{length}(\mathcal{F}) \wedge \mathcal{F}[i] \ \mathsf{instanceof} \ C \rightarrow \mathcal{H}(x_C)[\mathsf{f}((C)\mathcal{F}[i])] = \mathcal{F}(x_C)[i])$$

by the definition of $\lfloor . \rfloor$. Observe that the variables in $\bar{z}$ do not occur in heap_V. Therefore $(s, h)(s', h') \models$ (7.3) entails that $(s^*, h) \models \mathsf{heap}_V$. Formula (5.17) in heap_V implies that $(s^*, h) \models \mathsf{f}((C)\mathcal{H}[i]) = i$ whenever

$$(s^*, h) \models 0 \leq i < \mathsf{length}(\mathcal{H}) \wedge \mathcal{H}[i] \ \mathsf{instanceof} \ C \ .$$

Then $(s^*, h) \models \lfloor(7.2)\rfloor$ follows from $s^*(\mathcal{F}(x_C)) = s^*(\mathcal{H}(x_C))$, which completes our proof of $(s^*, h) \models \lfloor init[e_0, \bar{e}/\mathsf{this}, \bar{p}]\rfloor$.

In the third step we combine our previous results with our initial assumption $(s, h) \models$ (7.4). The carefully chosen values of $\bar{z}$ ensure that we have

$$(s^*, h) \models \mathsf{g}(init \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (Q''[\mathsf{rec}/\mathsf{this}][\mathsf{g}(.)/\mathsf{old}(.)]) \ . \qquad (7.5)$$

By (7.5) and the result of the second proof step we get

$$(s^*, h) \models Q''[\mathsf{rec}/\mathsf{this}][\mathsf{g}(.)/\mathsf{old}(.)] \ .$$

An application of Lemma 5.2 then yields $(s^*, h)(\mathsf{start}_{s^*}, \mathsf{start}_h) \models Q''[\mathsf{rec}/\mathsf{this}]$. And by Lemma 5.10 then $(s^*[\mathsf{this} \mapsto o], h)(\mathsf{start}_{s^*}, \mathsf{start}_h) \models Q''$, where the receiver $o = s^*(\mathsf{rec}) = \mathcal{E}[\![\lfloor e_0 \rfloor]\!](s, h)$. By expanding Q'' we find that our previous result means that there exists a sequence of values $\bar{\alpha}$ such that

$$(s_1, h)(\mathsf{start}_{s^*}, \mathsf{start}_h) \models (\mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}'] \ ,$$

where $s_1 = s^*[\mathsf{this} \mapsto o][\bar{z}' \mapsto \bar{\alpha}]$. Let $s_2 = s_1[\bar{u} \mapsto \bar{\alpha}]$. Our previous result also holds if we replace s_1 by s_2 because the local variables in $\bar{u}$ do not occur in the formula. By Lemma 5.10 then

$$(s_2, h)(\mathsf{start}_{s^*}, \mathsf{start}_h) \models \mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e \ .$$

Finally, let $s^\circ = s^*[\mathsf{this} \mapsto o][\bar{u} \mapsto \bar{\alpha}]$. We also have

$$(s^\circ, h)(\mathsf{start}_{s^*}, \mathsf{start}_h) \models \mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e \ . \qquad (7.6)$$

because the (fresh) variables in $\bar{z}'$ do not occur in this formula.

The first clause of the above formula implies that there exists a state (s_0, h_0) such that

$$\langle S, (s_0, h_0)\rangle \to (s^\circ, h) \tag{7.7}$$

and $(s_0, h_0)(\mathsf{start}_{s^*}, \mathsf{start}_h) \models \mathit{init}^+(\bar{p})$. So we have finished the third proof step.

Next, we want to use the computation in (7.7) to build a computation of the statement $u := e_0.m(e_1, \ldots, e_n)$. Let s_0' be the state that is obtained from s_0 by assigning the value of $s^*(v)$ to every local variable v that is no element of the set $\{\mathsf{this}, p_1, \ldots, p_n\}$. It is not difficult to prove that we also have

$$\langle S, (s_0', h_0)\rangle \to (s^\circ, h) \tag{7.8}$$

because a computation does not depend on the initial values of local variables other than the formal parameters and this. Moreover, it should be clear that we also have $(s_0', h_0)(\mathsf{start}_{s^*}, \mathsf{start}_h) \models \mathit{init}^+(\bar{p})$.

Let $v = \mathcal{E}[\![e]\!](s^\circ, h)$. We claim that we can derive the computation

$$\langle u := e_0.m(e_1, \ldots, e_n), (s^*, h_0)\rangle \to (s^*[u \mapsto v], h) \ , \tag{7.9}$$

from (7.8) using rule (MC_1). This requires us to prove that

$$\mathcal{E}[\![e_0]\!](s^*, h_0) = v_0 \notin \{\mathsf{null}, \bot\} \tag{7.10}$$

$$\mathcal{E}[\![e_i]\!](s^*, h_0) = v_i \neq \bot \quad \text{for } i \in \{1 \ldots n\} \tag{7.11}$$

$$s^*[\mathsf{this}, p_1, \ldots, p_n \mapsto v_0, v_1, \ldots, v_n] = s_0' \tag{7.12}$$

$$\mathcal{E}[\![e]\!](s^\circ, h) = v \neq \bot \tag{7.13}$$

Finally, if $v_0 = (E, i')$, then we should also prove that $\mathsf{meth}(E, m)$ (the implementation of m that is bound to calls on E-objects) is the implementation of method m in C.

In order to do so, we first show that the state (s^*, h_0) satisfies the conditions of Lemma 5.7. First, note that (7.8) guarantees that s° assigns the same values to logical variables as s_0'. By the construction of s° this also means that s_0' assigns the same variables to logical variables as s^*. Therefore $(s_0', h_0) \models \mathit{init}$ implies $(s^*, h_0) \models$ (5.27) by the construction of s^*. From $(s^*, h) \models \mathsf{heap}_V$ we get $(s^*, h_0) \models$ (5.28) because (5.28) does not depend on the heap. And finally, from $(s^*, h) \models \mathsf{heap}_V$ and $(s_0', h_0) \models \mathit{init}$ follows $(s^*, h_0) \models$ (5.29). Thus we have shown that all conditions of Lemma 5.7 are met.

We start with (7.10). We have

$$\begin{array}{ll}
\mathcal{E}[\![e_0]\!](s^*, h_0) & \{ \text{ Lemma 3.3 } \} \\
= \mathcal{N}[\![e_0]\!](s^*, h_0) & \{ \text{ Lemma 5.7, Lemma 5.6, (7.7) } \} \\
= \mathcal{N}[\![\lfloor e_0 \rfloor]\!](s^*, h) & \{ \text{ construction of } s^* \ \} \\
= \mathcal{N}[\![\lfloor e_0 \rfloor]\!](s, h) \ . &
\end{array}$$

The clauses $\lfloor \mathsf{defined}(e_0) \rfloor$ and $\lfloor \neg(e_0 = \mathsf{null}) \rfloor$ in $(s, h) \models$ (7.3) then ensure that $\mathcal{E}[\![e_0]\!](s^*, h_0) \notin \{\mathsf{null}, \bot\}$. Hence (7.10). Along the same lines we can show that (7.11) holds.

We have already observed that s'_0 and s^* assign the same values to logical variables. Moreover, by the construction of s'_0 they also agree on all local variables outside the set $\{\mathsf{this}, p_1, \ldots, p_n\}$. Hence the remaining cases of (7.12) that are yet to be proved are $\mathcal{E}[\![e_0]\!](s^*, h_0) = s'_0(\mathsf{this})$ and $\mathcal{E}[\![e_i]\!](s^*, h_0) = s'_0(p_i)$, for $i \in \{1 \ldots n\}$.

For the first case we have

$$\begin{array}{ll} \mathcal{E}[\![e_0]\!](s^*, h_0) & \{ \text{ see above } \} \\ = \mathcal{N}[\![\lfloor e_0 \rfloor]\!](s, h) & \{ \text{ construction of } s^* \ \} \\ = s^*(\mathsf{rec}) & \{ \text{ def. } o \ \} \\ = o & \{ \text{ construction of } s^\circ \ \} \\ = s^\circ(\mathsf{this}) & \{ \ (7.8) \ \} \\ = s'_0(\mathsf{this}) \ , & \end{array}$$

and for the second case we have

$$\begin{array}{ll} \mathcal{E}[\![e_i]\!](s^*, h_0) & \{ \text{ see above } \} \\ = \mathcal{N}[\![\lfloor e_i \rfloor]\!](s, h) & \{ \text{ def. } \mathsf{start}_{s^*} \text{ and } \mathcal{L}[\![\_]\!] \ \} \\ = \mathcal{L}[\![\mathsf{old}(p_i)]\!](s'_0, h_0)(\mathsf{start}_s, \mathsf{start}_h) & \{ \ (s'_0, h_0)(\mathsf{start}_s, \mathsf{start}_h) \models \mathit{init}^+(\bar{p}) \ \} \\ = s'_0(p_i) \ . & \end{array}$$

Hence (7.12) holds. The fourth condition (7.13) follows from the second clause of (7.6).

Finally, we must show that the call $e_0.m(e_1, \ldots, e_n)$ in state (s^*, h_0) is bound to the implementation in class C. We have $(s^*, h) \models \lfloor \mathsf{boundto}(e_0, C, m) \rfloor$ from (7.3), and by Lemma 5.7 then $(s^*, h_0) \models \mathsf{boundto}(e_0, C, m)$. If we expand the latter formula we get

$$(s^*, h_0) \models e_0 \ \mathsf{instanceof} \ C \wedge \bigwedge_{D \in \mathsf{overrides}(C)(m)} \neg(e_0 \ \mathsf{instanceof} \ D) \ ,$$

which implies that $C \preceq C$ (and therefore that E inherits the implementation of method m in class C), and moreover, E is not a subclasses of any of the classes that override this implementation. Hence $\mathsf{meth}(E, m)$ denotes the implementation of method m in class C, which completes our justification of (7.9).

The fifth step of our proof consists of showing that $(s^*, h_0)(s', h') \models Q$. Recall that we have $(s, h)(s', h') \models \lfloor Q \rfloor$, and therefore $(s^*, h)(s', h') \models \lfloor Q \rfloor$ by the construction of s^*. Hence $(s^*, h_0)(s', h') \models Q$ by Lemma 5.7.

In the sixth step we must prove that $(s^*[u \mapsto v], h)(s', h') \models Q'$. This follows immediately from the result of our previous step, the run in (7.9) and the validity of $\{Q\}u := e_0.m(e_1, \ldots, e_n)\{Q'\}$ using Def. 3.2.

We start the final step of our proof by repeating from Equation (7.6) that $(s^\circ, h) \models \mathsf{result} = e$. Moreover, $s^\circ(\mathsf{result}) = s^*(\mathsf{result})$ due to the construction of s°. These observations entail that

$$s^*[u \mapsto v](\mathsf{result}) = s^\circ[u \mapsto v](\mathsf{result}) = s^\circ(\mathsf{result}) = \mathcal{E}[\![e]\!](s^\circ, h) = v \ .$$

Applying Lemma 5.10 then yields $(s^*[u \mapsto v], h)(s', h') \models Q[\mathsf{result}/u]$. Note that u and all logical variables in $\bar{z}$ that obtained a different value in s^* do not occur in $Q[\mathsf{result}/u]$. For that reason we also have $(s, h)(s', h') \models Q[\mathsf{result}/u]$. □

7.3.6 Proof Outlines of Most General Method Specifications

In the previous section we have defined the Most General Method Specification (MGMS) of an arbitrary method, and we have shown that the MGMS enables us to prove every valid call specification. What remains to be done is to demonstrate that we can construct a valid proof outline for every MGMS. We will do so by proposing a particular annotation strategy for method bodies. Again, we will only consider the most difficult case which involves a method that returns a value.

Recall from Definition 7.9 that the MGMS of a method with formal parameters $\bar{p}$, body S and return value expression e consists of the precondition *init* and the postcondition $(\exists \bar{z}' \bullet (\mathsf{sp}(\mathit{init}^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}'])$. We propose the following Standard Annotation Pattern (SAP) for methods that are annotated with their MGMS.

Definition 7.10. *The Standard Annotation Pattern (SAP) of a method that returns a value is as follows.*

$$\begin{array}{l}\mathsf{requires}\ \mathit{init};\\ \mathsf{ensures}\ (\exists \bar{z}' \bullet (\mathsf{sp}(\mathit{init}^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}']);\\ t\ m(\bar{p})\ \{\ \ \mathsf{assert}\ \mathit{init}^+(\bar{p});\ S;\ \mathsf{assert}\ \mathsf{sp}(\mathit{init}^+(\bar{p}), S);\ \mathsf{return}\ e\ \ \}\end{array}$$

This annotation pattern has the following two verification conditions in our basic verification strategy (cf. Definition 7.2).

$$\mathit{init} \rightarrow \mathit{init}^+(\bar{p})[./\mathsf{old}(.)] \tag{7.14}$$

$$\mathsf{sp}(\mathit{init}^+(\bar{p}), S) \rightarrow (\mathsf{defined}(e) \rightarrow ((\exists \bar{z}' \bullet (\mathsf{sp}(\mathit{init}^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}'])[e/\mathsf{result}])) \tag{7.15}$$

The SAP is not a complete proof outline because it does not specify the annotation inside the body S. The SAP is only a valid proof outline if all verification conditions of the annotated method body S also hold. This set is defined by $VC(\mathit{init}^+(\bar{p}), S, \mathsf{sp}(\mathit{init}^+(\bar{p}), S))$. But first we show that both (7.14) and (7.15) trivially hold.

Lemma 7.9. *For every parameter sequence $\bar{p}$ we have $\models$ (7.14).*

Proof. Let $\bar{p} = p_1, \ldots, p_n$. Recall that $\mathit{init}^+(\bar{p}) \equiv \mathit{init} \wedge \bigwedge_{i=1}^{n}(p_i = \mathsf{old}(p_i))$. We have $\bigwedge_{i=1}^{n}(p_i = \mathsf{old}(p_i))[./\mathsf{old}(.)] \equiv \bigwedge_{i=1}^{n}(p_i = p_i)$, which is clearly a tautology. □

Lemma 7.10. *For every parameter sequence $\bar{p}$ and every expression e with $[e] \preceq [\mathsf{result}]$ we have $\models$ (7.15).*

Proof. Let $(s,h)(s',h') \models \mathsf{sp}(init^+(\bar{p}), S)$ and $(s,h)(s',h') \models \mathsf{defined}(e)$. The latter clause implies that $\mathcal{E}[\![e]\!](s,h) = v \neq \bot$. Moreover, from these clauses we conclude that $(s^*,h)(s',h') \models \mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e$, where $s^* = [\mathsf{result} \mapsto v]$, because result does neither occur in $\mathsf{sp}(init^+(\bar{p}), S)$ nor in e. Next, let $s^\dagger = s^*[\bar{z}' \mapsto s(\bar{u}')]$. Because the variables in $\bar{z}'$ are fresh we also have $(s^\dagger,h)(s',h') \models \mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e$. Repeated applications of Lemma 5.10 then yield

$$(s^\dagger,h)(s',h') \models (\mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}'] \ .$$

Hence $(s^\dagger,h)(s',h') \models (\exists \bar{z}' \bullet (\mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}'])$. This formula has no free occurrences of logical variables in $\bar{z}$, so we can replace $s^\dagger$ by s^*. The resulting formula is equivalent to

$$(s,h)(s',h') \models (\exists \bar{z}' \bullet (\mathsf{sp}(init^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e)[\bar{z}'/\bar{u}'])[e/\mathsf{result}]$$

according to Lemma 4.3. □

The task that remains is to show that we can construct a proof outline for S such that all verification conditions of $VC(init^+(\bar{p}), S, \mathsf{sp}(init^+(\bar{p}), S))$ hold. We will show that this is possible. In fact, we will define an annotation pattern that yields a valid proof outline for every valid Hoare triple provided that every relevant method is annotated with its MGMS. This annotation pattern is a function which for every valid Hoare triple $\{Q\}\ S\ \{Q'\}$ yields a valid proof outline for that triple. We call such a proof outline the Standard Proof Outline (SPO) of a Hoare triple, and we denote the SPO of a precondition Q, a statement S, and a postcondition Q' by $\mathsf{spo}(Q,S,Q')$. It is defined by induction on the structure of S. We discuss its most interesting case below; the full definition is listed in Figure 7.3.

The most difficult case of $\mathsf{spo}(Q,S,Q')$ concerns a while loop because there we have to come up with a suitable invariant. In the following, we use a solution to this problem that is described elsewhere by De Boer [dB91].

Intuitively, the invariant of a loop $\mathsf{while}\ (e)\ S$ must be a formula that holds in all final states of computations in which the body S has been executed a finite number of times. We only consider executions in which the guard e holds each time S is executed. Moreover, the initial state of each computation must satisfy precondition Q. Note that we cannot use the strongest postcondition $\mathsf{sp}(Q, \mathsf{while}\ (e)\ S)$ as invariant because it is too strong: it describes the final states of all *terminating* computations and thus excludes all partial computations of the loop.

Our weaker loop invariant is based on the strongest postcondition of Q and the augmented loop $\mathsf{while}\ (e \wedge u < u')\ S;\ u := u+1$. We assume here that u and u' are two fresh local variables. This augmented loop also terminates if the counter u reaches a particular limit u'. Thus its strongest postcondition also holds in the final states of partial computations of the original while loop. The complete loop invariant is

$$(\exists z : \mathsf{int} \bullet \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S;\ u := u+1)[z, z/u, u']) \ . \quad (7.16)$$

$$
\begin{aligned}
\mathsf{spo}(Q, t\ u, Q') &= t\ u \\
\mathsf{spo}(Q, u := e, Q') &= u := e \\
\mathsf{spo}(Q, e.x := e', Q') &= e.x := e' \\
\mathsf{spo}(Q, S_1; S_2, Q') &= \mathsf{spo}(Q, S_1, \mathsf{sp}(Q, S_1));\ \mathsf{assert}\ \mathsf{sp}(Q, S_1); \\
&\qquad\qquad \mathsf{spo}(\mathsf{sp}(Q, S_1), S_2, Q') \\
\mathsf{spo}(Q, u := \mathsf{new}\ C(\bar{e}), Q') &= \mathsf{intermediate}\ \mathsf{sp}(Q, u := \mathsf{alloc}(C));\ u := \mathsf{new}\ C(\bar{e}) \\
\mathsf{spo}(Q, e.m(\bar{e}), Q') &= e.m(\bar{e}) \\
\mathsf{spo}(Q, u := e.m(\bar{e}), Q') &= u := e.m(\bar{e}) \\
\mathsf{spo}(Q, \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2, Q') &= \mathsf{if}\ (e)\ \mathsf{spo}(Q \wedge e, S_1, Q') \\
&\qquad\qquad \mathsf{else}\ \mathsf{spo}(Q \wedge e = \mathsf{false}, S_2, Q') \\
\mathsf{spo}(Q, \mathsf{while}\ (e)\ S, Q') &= \mathsf{while}\ (e)\ \mathsf{assert}\ R;\ \mathsf{spo}(R \wedge e, S, R) \\
&\quad \text{where} \\
R &\equiv (\exists z : \mathsf{int} \bullet \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S;\ u := u + 1)[z, z/u, u'])
\end{aligned}
$$

Figure 7.3: The definition of $\mathsf{spo}(Q, S, Q')$. We assume in R that u, u' and z are variables of type int which do not occur in Q, S, or Q'.

Next, we will show by means of two lemmas that (7.16) satisfies the verification conditions of our basic verification strategy (Section 7.1.1).

Lemma 7.11. *For every statement S, every expression e, and every precondition Q in which u, u' and z do not occur we have $\models Q \rightarrow$ (7.16).*

Proof. Let $(s, h)(s', h') \models Q$. We will prove that

$$(s[z \mapsto 0], h)(s', h') \models \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1)[z, z/u, u'] \,. \tag{7.17}$$

Let $s^* = s[z, u, u' \mapsto 0, 0, 0]$. The claim in (7.17) is equivalent to

$$(s^*, h)(s', h') \models \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1)[z, z/u, u'] \tag{7.18}$$

because (7.18) does not depend on the values of u and u'. From Lemma 5.10 follows that (7.18) is equivalent to

$$(s^*, h)(s', h') \models \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1) \,. \tag{7.19}$$

Note that $(s^*, h)(s', h') \models Q \wedge u = 0$ follows from $(s, h)(s', h') \models Q$ by the construction of s^* and our assumption that u, u' and z do not occur in Q. Hence we can prove (7.19) by deriving

$$\langle \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1, (s^*, h) \rangle \rightarrow (s^*, h) \ ,$$

which is an easy task because $\mathcal{E}[\![e \wedge u < u']\!](s^*, h) = ff$. □

Lemma 7.12. *Let $\{Q\}$ while (e) S $\{Q'\}$ be a valid Hoare triple. Let u, u' and z be variables which do not occur in Q, Q', e and S. Then*

$$\models (7.16) \wedge e = \mathsf{false} \rightarrow Q' \ .$$

Proof. Let $(s,h)(s',h') \models (7.16) \wedge e = \mathsf{false}$. Then there exists an $\alpha \in dom(\mathsf{int})$ such that

$$(s[z \mapsto \alpha], h)(s', h') \models \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1)[z, z/u, u'] \ . \tag{7.20}$$

Let $s^* = s[z, u, u' \mapsto \alpha, \alpha, \alpha]$. From (7.20) follows

$$(s^*, h)(s', h') \models \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1)[z, z/u, u'] \tag{7.21}$$

because the formula in (7.20) does not depend on the values of u and u'. By means of Lemma 5.10 we then get

$$(s^*, h)(s', h') \models \mathsf{sp}(Q \wedge u = 0, \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1) \ . \tag{7.22}$$

So there exists a state (s_0, h_0) such that $(s_0, h_0)(s', h') \models Q$, $s_0(u) = 0$, and $\langle \mathsf{while}\ (e \wedge u < u')\ S; u := u + 1, (s_0, h_0)\rangle \to (s^*, h)$. We must prove that $(s,h)(s',h') \models Q'$, which is equivalent to

$$(s^*[u \mapsto 0], h)(s', h') \models Q' \tag{7.23}$$

because u, u' and z do not occur in Q'. It is not difficult to see that the existence of the run of the augmented loop and $(s^*, h)(s', h') \models e = \mathsf{false}$ imply the existence of the computation $\langle \mathsf{while}\ (e)\ S, (s_0, h_0)\rangle \to (s^*[u \mapsto 0], h)$. Then (7.23) follows from $\{Q\}\ \mathsf{while}\ (e)\ S\ \{Q'\}$. □

We must also prove that (7.16) is a real invariant of the original loop.

Lemma 7.13. *Let* $\mathsf{while}\ (e)\ S$ *be a valid statement, and let* u, u' *and* z *be variables which do not occur in* Q, e *and* S. *Then*

$$\models \{(7.16) \wedge e\}\ \mathsf{while}\ (e)\ S\ \{(7.16)\} \ .$$

Proof Sketch. We only give a sketch of the proof because the proof itself is rather straightforward. Note that we can extend every terminating execution of the body S with an execution of $u := u + 1$. If we append this computation to the computation of the augmented loop whose existence is ensured by the precondition, then we get the terminating computation whose existence is claimed in the postcondition. If the first computation has n executions of the body, then we can prove the postcondition for $z = n + 1$. □

Theorem 7.14. *For all valid Hoare triples* $\{Q\}S\{Q'\}$ *we have* $\models Q''$ *for every* $Q'' \in VC(Q, \mathsf{spo}(Q, S, Q'), Q')$ *if every method in the program is annotated with its MGMS.*

Proof. By structural induction on S. The cases where $S \equiv u := e$, $S \equiv t\ u$, and $S \equiv e.x := e'$ are covered by Theorem 4.5, Theorem 4.7, and Theorem 4.12, respectively. For $S \equiv S_1; S_2$ we first prove that $\models \{Q\}S_1; S_2\{Q'\}$ implies $\models \{\mathsf{sp}(Q, S_1)\}S_2\{Q'\}$. Then this case follows from the induction hypothesis

and Theorem 7.7. For $S \equiv \mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2$ we must prove that the validity of $\{Q\}\mathsf{if}\ (e)\ S_1\ \mathsf{else}\ S_2\{Q'\}$ implies $\models\{Q \wedge e\}S_1\{Q'\}$ and $\models\{Q \wedge e = \mathsf{false}\}S_2\{Q'\}$. Then this case follows also from the induction hypothesis. For $S \equiv u := e.m(\bar{e})$ we can simply refer to Theorem 7.8. We can prove a similar (and slightly simpler) result for $S \equiv e.m(\bar{e})$. For the case $S \equiv u := \mathsf{new}\ C(\bar{e})$ we must prove the validity of two verification conditions. The first one corresponds to the allocation step and is covered by Lemma 6.12 and Theorem 7.7. The validity of the verification condition that corresponds to the call to the constructor method can be proved along the lines of the proof of Theorem 7.8, but this case is less complex because it does not involve dynamic binding. Finally, for $S \equiv \mathsf{while}\ (e)\ S'$ we can prove our claim using the induction hypothesis, and the Lemmas 7.11-7.13. □

With this result we can return to the Standard Annotation Pattern of Definition 7.10. If we replace the body S in Definition 7.10 by the proof outline $\mathsf{spo}(\mathit{init}^+(\bar{p}), S, \mathsf{sp}(\mathit{init}^+(\bar{p}), S))$ then we get a valid proof outline for an arbitrary method that is annotated with its Most General Method Specification.

Theorem 7.15. *Every verification condition of the basic verification strategy (Section 7.1.1) for an arbitrary method with the following annotation holds.*

$$
\begin{array}{l}
\mathsf{requires}\ \mathit{init};\\
\mathsf{ensures}\ \mathsf{sp}(\mathit{init}^+(\bar{p}), S) \wedge \mathsf{defined}(e) \wedge \mathsf{result} = e;\\
t\ m(\bar{p})\ \{\\
\quad \mathsf{assert}\ \mathit{init}^+(\bar{p});\\
\quad \mathsf{spo}(\mathit{init}^+(\bar{p}), S, \mathsf{sp}(\mathit{init}^+(\bar{p}), S));\\
\quad \mathsf{assert}\ \mathsf{sp}(\mathit{init}^+(\bar{p}), S);\\
\quad \mathsf{return}\ e\\
\}
\end{array}
$$

Proof. By Theorem 7.7 we have $\models\{\mathit{init}^+(\bar{p})\}S\{\mathsf{sp}(\mathit{init}^+(\bar{p}), S)\}$. Then we use Theorem 7.14 to show that the verification conditions of the annotated method body $\mathsf{spo}(\mathit{init}^+(\bar{p}), S, \mathsf{sp}(\mathit{init}^+(\bar{p}), S))$ hold. The validity of the two remaining verification conditions follows from Lemma 7.9 and Lemma 7.10. □

For other method types we can prove similar results. Thus we have shown that our proof outline logic is relatively complete.

Chapter 8

Modularity

The proof outline logic that we have developed in the preceding chapters of this thesis takes *closed* programs as input. That is, we have always assumed that all classes of the program are available, and we have not yet considered the question what happens to a proof when new classes are later added to an existing program. We will endeavor to provide an analysis of the consequences of this type of program extension in this chapter.

There are at least two significant reasons for considering this question. The first reason is that it addresses a reality in object-oriented software engineering. Almost all object-oriented software reuses prefabricated classes in standard *modules* (or *packages*) which are supplied by other parties. Programs that are written in this way can be seen as extensions of the programs formed by these standard classes. Ideally, such standard classes are shipped with formal specifications, which can then be used to reason about calls to methods in these classes without having to see their actual implementation.

Another reason for considering this question is that it may lead to a proof system which supports division of labor. If a program consists of several parts which are merged at some later stage, then we would like to know whether verified specifications of the individual parts still hold. The object-oriented development process described in the previous paragraph is but one example of division of labor; software maintenance can also be seen as a form of division of labor. Adequate support for division of labor clearly contributes to the scalability of a proof method.

Program logics for reasoning about open, extensible programs are usually called *modular*. However, this term is not always adequately defined. One notable definition has been given by Leavens and Weihl [LW90]: '... specification and verification techniques must be *modular* in the sense that when new types of objects are added to a program, unchanged program modules should not have to be respecified or reverified'. The expression 'types of objects' can be replaced by 'classes' in Leavens and Weihl's description of modularity. We will use the adjective *modular* according to this definition.

We would like to emphasize that there are two distinct problems which may both compel us to respecify a program module. Firstly, a given specification may no longer be *accurate*. In other words, we could be forced to replace an existing specification because it is no longer valid. However, it may also be the case that a specification is still accurate, but no longer *adequate* in the sense that it has become too weak. For example, it could be the case that a method specification describes the behavior of a method for arguments of specific types, but that it does not tells us what happens if we pass it an argument of some new subtype. The first issue has received a considerable amount of attention in the literature, but a definite solution for the second problem is still missing. We will investigate both matters in this chapter.

This chapter is organized as follows. We start with a section that provides a formal account of *behavioral subtyping* in the context of annotated cOOre programs. Behavioral subtyping is a kind of subtyping that requires objects of subtypes to behave in the same way as the objects of their supertype [Ame91, LW94]. The main result of that section is a static test (a specification match) for determining whether a subclass is a behavioral subtype of its superclass. This test is based on the same techniques as our object-oriented adaptation rules.

In Section 8.2 we develop two new adaptation rules which turn our proof outline logic into a modular proof system. The first rule is based on the well-known supertype abstraction principle [LW90, LW95]. We will argue that this principle leads to program logics that are incomplete. Subsequently, we will define a new principe called *subtype awareness* and develop the corresponding, stronger adaptation rule.

In Section 8.3 we define a new completeness notion called behavioral modular completeness. We propose this new completeness notion because the standard relative completeness notion is too stringent for program logics that are based on behavioral subtyping [PdB05a].

In the last section of this chapter we first introduce an open reasoning problem for modular program logics. Next we evaluate several advanced specification constructs that may lead to a solution for this problem.

8.1 Behavioral Subtyping

Recall from Section 2.1 that each subclass *inherits* the fields and methods of its superclass. Thus inheritance proceeds along the *subclass* relation. By contrast, the *subtype* relation essentially determines whether the objects of some type t can 'play the role' of objects of type t'. Recall that the typing rules of the language allow objects of subtypes to enter the domains of their supertypes. For example, an assignment $u := e$ is also permitted if the type of e is some subtype of the type of u. Naturally, it is important that no unexpected behavior results from this design decision.

In the preceding chapters of this thesis we have always equated the subtype

relation with the subclass relation, as is done in most object-oriented programming languages. This couples inheritance (which proceeds along the *subclass* relation) with subtyping. However, several people have opposed this equation. For example, Snyder [Sny86] writes that 'subtyping should not be tied to inheritance'. And America [Ame87] argues that 'inheritance and subtyping should be decoupled and that each should be considered in its own right'.

The main objection against equating the subtype relation with the subclass relation is that inheritance *only* ensures that a subclass has the same internal structure (fields). But it does *not* ensure that objects of a subclass behave in the same way as objects of their superclass. For a subclass can potentially change the behavior of inherited methods by overriding them.

Behavioral subtyping can be seen as a less liberal form of subtyping. Having the same internal structure or external structure (method signatures) is not enough when it comes to behavioral subtyping. It also demands that what holds regarding the *behavior* of objects of a particular type also holds for all objects of its subtypes [Ame91, LW94]. We will give a more formal definition of behavioral subtyping in Section 8.1.3.

Forcing a program to satisfy the *behavioral* subtype relation has important benefits for proof systems. The following section reveals by means of an example what goes wrong if we adopt a weaker definition of subtyping.

8.1.1 Class Addition May Invalidate Proofs

We demonstrate in this section that program extensions can render method specifications invalid. The following example also sheds light on the benefits of behavioral subtyping.

Consider the following (closed) COORE program $\pi \equiv$ Client One, where

$$\begin{array}{rcl} \mathsf{Client} & \equiv & \mathsf{class}\ \mathit{Client}\ \{\ \mathsf{void}\ \mathit{initOne}(\mathit{One}\ o)\{\ o.\mathit{init}()\ \}\ \} \\ \mathsf{One} & \equiv & \mathsf{class}\ \mathit{One}\ \{\ \mathsf{int}\ x;\ \mathsf{void}\ \mathit{init}()\ \{\ \mathsf{this}.x := 1\ \}\ \}\ . \end{array}$$

The constructor methods of both classes are irrelevant in our example and are therefore omitted. It is not difficult to see that the following specification holds for method *initOne* of class *Client* in π.

$$\{\mathsf{true}\}\ \mathit{initOne}@\mathit{Client}\ \{\mathsf{old}(o).x = 1\} \tag{8.1}$$

That is, field x of the object that is passed to the *initOne* method is set to 1.

Next, consider the program $\pi' \equiv$ Client One Two that is obtained by extending π with class *Two* below.

$$\mathsf{Two} \quad \equiv \quad \mathsf{class}\ \mathit{Two}\ \mathsf{extends}\ \mathit{One}\ \{\ \mathsf{void}\ \mathit{init}()\ \{\ \mathsf{this}.x := 2\ \}\ \}$$

Class *Two* inherits field x of class *One*. Thus it has the same internal structure as class *One*. Its external structure is also equal to that of class *One* because it provides the same methods. But the behavior of objects of class *Two* differs from that of objects of class *One* because the behavior of the overriding

implementation of method *init* in class *Two* is different. So class *Two* is not a behavioral subtype of class *One*.

Does the addition of class *Two* have consequences for the validity of properties of the original program π? It is not difficult to see that specification (8.1) no longer holds. Consider what happens when method *initOne* is passed an instance of class *Two*. This is permitted if *Two* is a subtype of *One*. The call $o.init()$ is then bound to the implementation in class *Two*, which invalidates the postcondition of (8.1). Thus the example reveals that class addition may invalidate proofs.

A way to prevent this scenario is to partially decouple the subtype relation from the subclass relation by insisting that every subtype must be a behavioral subtype. This would forbid clients to call the *initOne* method with an actual parameter of type *Two*.

8.1.2 A Modular Programming Language

We have seen in the previous section that behavioral subtyping has important advantages over ordinary subtyping for modular reasoning. But we have also seen that not every subclass is a behavioral subtype. For these reasons we will develop a *modular* variant of cOOre in this section, dubbed m-cOOre, which partially decouples the subtype relation from the subclass relation. More precisely, the subtype relation in m-cOOre will be a subset of the subclass relation. This gives programmers the opportunity to reuse code by means of inheritance without the obligation that *every* new subclass must behave in the same way as its superclass.

A minimal requirement for subtypes is that they have the same fields and methods as their supertypes. Our decision to view the subtype relation as a subset of the subclass relation means that every subtype will also be a subclass of its supertype. Hence it will automatically inherit the required fields and methods from its supertype.

It is possible to further decouple the subtype relation from the subclass relation. This could be done by dropping the requirement that each subtype must be a subclass of its supertype. The compiler would then have to check whether each subtype declares the same fields and methods as its supertype. This is similar to a class implementing an *interface* in Java [GJSB00] or C# [Mok03]. It would open up the possibility to declare several supertypes for one class. However, we do not think that this situation requires interesting additional reasoning techniques. Therefore we choose to work out the simpler situation in which each subtype is a subclass of its supertype.

To accommodate the proposed functionality of m-cOOre we define the grammar of the syntax of its class definitions in the following way.

$$\begin{array}{lll} \mathit{class} \in \mathit{Class} & ::= & \mathsf{class}\ C\ (\epsilon \mid \mathsf{extends}\ D)\ (\epsilon \mid \mathsf{refines}\ E) \\ & & \mathsf{imports}\ \bar{C}\ \{\ \mathit{field}^*\ \mathit{constr}\ \mathit{meth}^*\} \end{array}$$

In other words, the syntax of a class in M-COORE additionally contains a refines clause and an imports clause. The syntax of M-COORE equals the syntax of COORE (see Fig. 2.1) in all other respects.

A clause extends D continues to define the subclass relation. The new clause refines E indicates that C is a subtype of E. Such a clause will lead to new proof obligations for class C which will only hold if C is indeed a behavioral subtype of E. We additionally require that class E is a superclass of class D, thus ensuring that the subclass relation subsumes the subtype relation. An omitted refines clause is equivalent to refines D. We write $C \prec E$ if C is declared as a subtype of E. The subtype relation $\preceq$ in M-COORE is the reflexive and transitive closure of the $\prec$ relation.

To obtain a modular proof system for M-COORE programs we have to analyze on which classes a particular class *depends*. We say that a class C explicitly depends on some class D when the corresponding type D occurs in the code of C (e.g., as the static type of a field). The classes on which C explicitly depends form the *type range* of C. A clause imports $\bar{C}$ explicitly lists the type range of C. By $\bar{C}$ we denote a comma-separated list $C_1, \ldots, C_l$ of class names.

Only class names that are listed in the type range $\bar{C}$ may occur in the class implementation. Moreover, both D and E (or their default values if the clauses are omitted) should occur in $\bar{C}$. The class name C is *not* allowed to occur in $\bar{C}$ but may nevertheless be used in its implementation. We will assume that, for every class C, $\mathsf{impcl}(C)$ denotes the set of classes that C imports.

Class Scopes

While we restrict all use of class names in the implementation of a class to class names that occur in the imports list, we have to be more liberal when it comes to specifications. For a similar restriction would easily lead to inadequate specifications, as the following example shows.

Example 8.1. *Suppose that we have three classes C, D, and E such that C imports D, and D imports E. Furthermore, assume that class D has a method m that allocates an object of class E. Now consider a method m' in C that calls method m in D. After execution of m' we know that one object of class E has been added to the heap. But how can we express that in the specification of method m' without mentioning its type E?*

This example suggests that we must allow specifications to refer to all class names in the transitive closure of the imports relation of a class C. We will call this set the *scope* of C and denote it by $\mathsf{scope}(C)$. Formally, $\mathsf{scope}(C)$ is defined as the least set of classes such that $D \in \mathsf{scope}(C)$ if one of the following holds.

i) $D \equiv C$
ii) $D \in \mathsf{impcl}(C)$
iii) there exists a class E such that $E \in \mathsf{scope}(C)$ and $D \in \mathsf{scope}(E)$

The scope of a class also limits the set of types that expressions in that class may have.

Lemma 8.1. *For every well-typed expression e in the context of a class C we have* $[e] \in \mathsf{scope}(C) \cup \{\mathsf{undefT}, \mathsf{NullT}, \mathsf{int}, \mathsf{boolean}\}$.

Proof. By structural induction on e. The only complex case concerns an expression of the form $e'.x$. Let field x be defined in class D. Then $[e'.x] \in \mathsf{impcl}(D)$, and consequently $[e'.x] \in \mathsf{scope}(D)$. From the type rules follows that $[e']$ is a subclass of D. Next, observe that the scope of a class subsumes its superclasses. So we also have $D \in \mathsf{scope}([e'])$. From this result and $[e'.x] \in \mathsf{scope}(D)$ follows $[e'.x] \in \mathsf{scope}([e'])$. Furthermore, by the induction hypothesis we have $[e'] \in \mathsf{scope}(C)$ because $[e']$ must be a class name. Hence $[e'.x] \in \mathsf{scope}(C)$. □

8.1.3 Behavioral Subtyping and Specification Matching

According to Leavens and Dhara, behavioral subtyping is essentially refinement of object types [LD00]. We will give a formal definition of behavioral refinement in this section and subsequently define behavioral subtyping in terms of behavioral refinement. This definition of behavioral subtyping raises the question how one can check whether a particular type is a behavioral refinement of some other type. We will therefore also look at specification matching [CC00] in this section, which is a static way to prove behavioral refinement on the basis of method specifications.

Behavioral Refinement

So far, we have said that behavioral subtyping requires that objects of subtypes behave in the same way as objects of their supertypes. The behavior of an object is determined by the behavior of methods called on the object. We therefore have to investigate the behavior of methods in subtypes. In particular, we must focus on the behavior of *overriding* methods. For method implementations which are simply inherited (unchanged) from the supertype will not cause objects of the subtype to behave differently. And functionality in subtypes which is not present in supertypes is also irrelevant because one cannot ask objects of the supertype to perform such functions.

For clarity, we first summarize our definition of method overriding as given in Chapter 2.

Definition 8.1. *A method m in some class C with return type t and parameter types $t_1, \ldots, t_n$ overrides a method m' in class D with return type t' and parameter types $t'_1, \ldots, t'_n$ if and only if the following conditions hold.*

i) $m \equiv m'$
ii) class C is a subclass of class D
iii) $t'_i \preceq t_i$ *for every* $i \in \{1 \ldots n\}$
iv) $t \preceq t'$.

The last clause is dropped if both methods have return type void. Constructor methods are never inherited and can therefore not be overridden. Observe that all these conditions can be checked statically.

For simplicity, we will only allow a method in a subclass to have the same name as a method in one of its superclasses if the method in the subclass overrides the other method. This ensures that we do not have to deal with method overloading, which would needlessly complicate our definitions.

We are now ready to define behavioral refinement for methods. We assume that all methods are annotated with a precondition and a postcondition. A method essentially refines the behavior of another method that it overrides when it satisfies the specification of the overridden method. However, it only has to satisfy this specification for parameter values that are also accepted by the overridden method. This is sufficient to ensure that calls to the overridden method that are handled by the overriding method result in similar behavior.

Definition 8.2. *Let m be a method in class C with formal parameters $\bar{p} = p_1, \ldots, p_n$ and return type t. Assume that m overrides a method m' in class D with parameters $\bar{p}' = p'_1, \ldots, p'_n$ and return type t'. We say that m refines the behavior of m' if and only if m satisfies the interface specification with precondition (8.2) and postcondition (8.3) below, where P' and Q' are the precondition and postcondition of m', respectively, and $I = \{i \in \{1 \ldots n\} | [p_i] \preceq \mathsf{object}\}$.*

$$P'[\bar{p}/\bar{p}'] \wedge \bigwedge_{i \in I} (p_i \ \mathsf{instanceof} \ [p'_i]) \tag{8.2}$$

$$Q'[\mathsf{old}(.[\bar{p}/\bar{p}'])/\mathsf{old}(.)] \tag{8.3}$$

The substitution $[\bar{p}/\bar{p}']$ in (8.2) replaces every parameter in $\bar{p}'$ by the corresponding parameter in $\bar{p}$. Note that the restrictions on the parameter types of overriding methods reveal that this substitution may replace an expression of a subtype by an expression of a supertype (see Definition 8.1). The potentially 'weaker' types of the parameters in $\bar{p}$ may contain less information about the types of the parameters. The second clause of the precondition restores the lost information.

The postcondition Q' may also contain occurrences of formal parameters inside expressions of the form $\mathsf{old}(e)$. The function $[\mathsf{old}(.[\bar{p}/\bar{p}'])/\mathsf{old}(.)]$ replaces these occurrences by the corresponding formal parameters of the overriding method. Its characteristic case is

$$\mathsf{old}(e)[\mathsf{old}(.[\bar{p}/\bar{p}'])/\mathsf{old}(.)] \equiv \mathsf{old}(e[\bar{p}/\bar{p}']) \ .$$

The definition of $[\mathsf{old}(.[\bar{p}/\bar{p}'])/\mathsf{old}(.)]$ is extended to all other logical expressions and formulas in the standard way.

Finally, we have to make a remark about the types of this and result in these formulas. We assume that this and result in (8.2) and (8.3) retain their original types. That is, we have $[\mathsf{this}] = D$ and $[\mathsf{result}] = t'$ in these formulas. Changing the types of these variables to the types which they have in the context of

method m might change the meanings of (8.2) and (8.3) in the presence of field shadowing. One can simply think of occurrences of this and result as being (implicitly) annotated with a specific type.

The following lemmas provide semantical characterizations of the parameter renaming operations $[\bar{p}/\bar{p}']$ and $[\mathsf{old}(.[\bar{p}/\bar{p}'])/\mathsf{old}(.)]$ in (8.2) and (8.3), respectively. They basically say that parameter renaming has no influence on the validity of assertions if we ensure that the parameters have the same values.

Lemma 8.2. *Let $\bar{p} = p_1, \ldots, p_n$ and $\bar{p}' = p'_1, \ldots, p'_n$ be two parameter sequences such that $[p'_i] \preceq [p_i]$ for every $i \in \{1 \ldots n\}$, and let $\bar{v} = v_1, \ldots, v_n$ be a sequence of values such that $v_i \in dom([p'_i])$, for every $i \in \{1 \ldots n\}$. Let P be an arbitrary precondition. Then we have for every valid state $(s[\bar{p} \mapsto \bar{v}], h)$ that*

$$(s[\bar{p}' \mapsto \bar{v}], h) \models P \quad \Longleftrightarrow \quad (s[\bar{p} \mapsto \bar{v}], h) \models P[\bar{p}/\bar{p}'] \wedge \bigwedge\nolimits_{i \in I} (p_i \ \mathsf{instanceof}\ [p'_i]) \ ,$$

where $I = \{i \in \{1 \ldots n\} | [p_i] \preceq \mathsf{object}\}$ and $s[\bar{p} \mapsto \bar{v}]$ denotes the stack that results from s by assigning v_i to p_i, for every $i \in \{1 \ldots n\}$.

Proof. By structural induction on P. □

Lemma 8.3. *Let $\bar{p}$, $\bar{p}'$ and $\bar{v}$ be as in Lemma 8.2. Let Q be an arbitrary assertion. Then we have, for every state (s, h) and every compatible (and valid) freeze state $(s'[\bar{p}' \mapsto \bar{v}], h')$, that*

$$(s, h)(s'[\bar{p}' \mapsto \bar{v}], h') \models Q \quad \Longleftrightarrow \quad (s, h)(s'[\bar{p} \mapsto \bar{v}], h') \models Q[\mathsf{old}(.[\bar{p}/\bar{p}'])/\mathsf{old}(.)] \ ,$$

where $s[\bar{p} \mapsto \bar{v}]$ denotes the stack that results from s by assigning v_i to p_i, for every $i \in \{1 \ldots n\}$.

Proof. By structural induction on Q. □

Behavioral Subtyping

With the definition of behavioral refinement above we can give a formal definition of behavioral subtyping. To be precise, we will define under what conditions two types C and D (both object types) are behavioral subtypes. Roughly speaking, two types are behavioral subtypes if every overriding method of the subtype refines the behavior of the overridden method of the supertype.

In the following definition the expression 'a method of class C' either denotes a method implementation declared in class C *or* a method implementation which C inherits and which is not overridden by another method of C.

Definition 8.3. *A class C is a behavioral subtype of a class D if*

i) C is a subclass of D, and

ii) every method m of C which overrides a method m' of D refines the behavior of m'.

Thus behavioral subtyping restricts the behavior of methods in subtypes. It should be clear that not every class that is declared as a subtype of some other class in an arbitrary M-COORE program automatically qualifies as a proper behavioral subtype. We must carefully check whether the subtype relation of a program satisfies the criteria of behavioral subtyping.

Specification Matching

Ideally, we would like to be able to infer on the basis of its specification whether a given method behaviorally refines some other method. More specifically, we would like to have a function that tells us whether two specifications 'match' in the sense that every method that satisfies the first specification refines the behavior of the second specification. Applying such a function is known as *specification matching*. Specification matching can be used, for example, to determine whether one can replace a particular component in a system by some other component without changing the system's observable behavior (cf. [CC00]).

It is customary to use logical formulas over the elements of the two specifications (called specification matches) as match criteria. Chen and Cheng have compiled an overview of specification matches which contains no less than ten entries [CC00]. However, these specification matches are only suitable for simple languages with global variables. We will develop an object-oriented specification match below.

Our search for an object-oriented specification match was simplified by the observation that there is a surprising similarity between specification matches and the verification conditions of adaptation rules (cf. Section 5.2.1). The verification condition of an adaptation rule checks whether the specification of a method call follows from the specification of the corresponding method, whereas a specification match checks whether the specification of a particular method implies the specification of the method that it is assumed to refine. This is almost the same problem!

This similarity enables us to use the verification conditions of our adaptation rules as blueprints for our object-oriented specification match. We will show that we get a proper object-oriented specification match by simply translating the verification condition (*VC5*) of our basic adaptation rule (on p. 80) to the specification matching context. The resulting specification match is even simpler than (*VC5*) because it neither involves parameter passing nor result value handling. However, we do have to handle differences in parameter sets between the two methods.

Let $\{P\}\ m@C\ \{Q\}$ be the interface specification of a method m in class C with formal parameters $\bar{p} = p_1, \ldots, p_n$. Let m' be a method in some class D with specification $\{P'\}\ m'@D\ \{Q'\}$ and formal parameters $\bar{p}' = p'_1, \ldots, p'_n$. Our specification match for these two methods is the following formula.

$$\mathsf{heap}_V^D \wedge \mathsf{g}'(P') \wedge \bigwedge_{i\in I}(p_i \text{ instanceof } [p_i'])$$
$$\wedge\, (\forall \bar{z} \in \mathcal{H} \bullet \lfloor P \rfloor \rightarrow (Q[\lfloor . \rfloor / \mathsf{old}(.)])) \rightarrow (Q'[\mathsf{g}'(.)/\mathsf{old}(.)]) \quad (8.4)$$

The predicate heap_V^D in (8.4) abbreviates the conjunction of heap^D (to be explained below) and the formula $\bigwedge_{v\in V} v \in \mathcal{H}$, where V is the set that contains this, and every parameter in $\bar{p}$ that references an object, and every logical variable z that occurs free in P' or Q' which either references an object or a sequence of objects. (We have $v \sqsubseteq \mathcal{H}$ instead of $v \in \mathcal{H}$ for every logical variable v that references a sequence of objects.) Thus this clause says that all these variables reference objects in the initial heap $\mathcal{H}$.

The formula heap^D is a restricted version of the formula heap (as defined on p. 75). It is obtained from heap by restricting the clauses (5.19) and (5.20) to fields which are defined in classes that are part of the scope of class D. This restriction makes sense because we do not want (8.4) to depend on fields that are not visible in the context of the overriding method.

The list $\bar{z}$ in (8.4) contains all logical variables that occur free in P or Q (except the special-purpose logical variable result that denotes the result value). The specification match is evaluated in the context of the overriding method. However, we assume again that this has no consequences for the types of occurrences of this and result in P' and Q'.

Other new elements of (8.4) are the function g' and the operation $[\lfloor . \rfloor / \mathsf{old}(.)]$. All other elements of (8.4) are defined in the same way as in Definition 8.2. The function g' combines the substitution operation $[\bar{p}/\bar{p}']$ and $\lfloor . \rfloor$. We have $\mathsf{g}'(Q) \equiv \lfloor Q[\bar{p}/\bar{p}'] \rfloor$.

The operation $[\lfloor . \rfloor / \mathsf{old}(.)]$ simply replaces every expression of the form $\mathsf{old}(e)$ in a formula by the corresponding expression $\lfloor e \rfloor$. It is a simplified version of the corresponding operation $[\mathsf{g}'(.)/\mathsf{old}(.)]$ in (*VC5*), which also takes the context switch into account. Its effect is described by the following lemma.

Lemma 8.4. *Let e be an arbitrary expression and let Q be an arbitrary assertion. Let $\mathcal{H}$ be a fresh logical variable of type* object**. Let $\mathcal{H}(x_C)$ be a fresh logical variable of type t^*, for every instance variable $x : t$ declared in some class C which occurs in e or Q. Let h, h' be two heaps such that $dom(h) \subseteq dom(h')$. Let s be a stack such that $(s, h) \models (5.28)$ and $(s, h) \models (5.29)$. Then*

i) $\mathcal{E}[\![e]\!](s, h) = \mathcal{N}[\![\lfloor e \rfloor]\!](s, h')$, and

ii) $(s, h')(s, h) \models Q$ if and only if $(s, h') \models (Q[\lfloor . \rfloor / \mathsf{old}(.)])$.

Proof. Along the lines of the proof of Lemma 5.7. □

The following example shows how one can use our specification match to check whether a type *CachedAccount* is a behavioral subtype of a type *Account*.

Example 8.2. *Assume that we have a class Account that models a bank account. It has a field bal* : int *that represents the account's current balance. Moreover,*

it has a method setBalance(int p) *that sets the account's balance to a new value. We assume that the following specification holds.*

$$\{\mathsf{true}\}\mathit{setBalance}@\mathit{Account}\{\mathsf{this}.\mathit{bal} = \mathsf{old}(p)\}$$

Next, we also assume that we have a subclass CachedAccount of Account which stores the previous balance value in a field cache : int. *It overrides method setBalance in Account with a method implementation which satisfies*

$$\{\mathsf{true}\}\mathit{setBalance}@\mathit{CachedAccount}\{\mathsf{this}.\mathit{bal} = \mathsf{old}(p) \wedge \mathsf{this}.\mathit{cache} = \mathsf{old}(\mathsf{this}.\mathit{bal})\}\ .$$

Is CachedAccount a behavioral subtype of Account? The specification match for the implementation of setBalance in CachedAccount (without the clause $\mathsf{heap}_V^{\mathit{CachedAccount}}$*) is*

$$\mathsf{true} \wedge (\mathsf{true} \rightarrow \mathsf{this}.\mathit{bal} = p \wedge \mathsf{this}.\mathit{cache} = \mathcal{H}(\mathit{bal}_{\mathit{Account}})[\mathsf{f}(\mathsf{this})]) \rightarrow \mathsf{this}.\mathit{bal} = p$$

It is not difficult to see that this formula holds. So CachedAccount is indeed a behavioral subtype of Account.

The specification match that we propose here is better than the specification match that is used in early work on behavioral subtyping [Ame91, LW94] because it identifies more matches. This fact follows from Chen and Cheng's analysis of existing specification matches. Our specification match is essentially an object-oriented version of their $M_{gen-pred}$ match [CC00, p. 93], whereas traditionally the $M_{plug-in}$ match is used. Chen and Cheng prefer a specification match which has one additional clause (see also [DL96]), but this is explained by the fact that they consider specifications for *total* correctness, whereas we investigate partial correctness.

The following theorem justifies our specification match by showing that it implies behavioral refinement.

Theorem 8.5. *Let m be a method in some class C with formal parameters $\bar{p} = p_1, \ldots, p_n$, and let m' be a method in class D with formal parameters $\bar{p}' = p'_1, \ldots, p'_n$, precondition P' and postcondition Q'. Suppose that m overrides m', and that $\models \{P\}\ m@C\ \{Q\}$. Then $\models$ (8.4) implies that m refines the behavior of m'.*

Proof. Let S be the body of the implementation of method m, and let e be the expression that denotes its result value. Let $\langle S, (s,h)\rangle \rightarrow (s',h')$ be such that $\mathcal{E}[\![e]\!](s,h) = v \neq \bot$, and let $(s,h) \models (8.2)$. We must prove that

$$(s'[\mathsf{result} \mapsto v], h')(s,h) \models (8.3)\ . \tag{8.5}$$

Many steps in this proof are equal to steps in the soundness proof of the adaptation rule (cf. Theorem 5.11). In fact, the proof below is simpler because

the parameter renaming operations in (8.4) are simpler then the context switch in Theorem 5.11. Moreover, (8.4) does not have to handle result value passing. We will nevertheless give the proof here for completeness sake.

Our first step is (again) to build a stack in which the variables of the heap model have the same values as the corresponding locations in the initial heap h. Let $\bar{v}$ be an arbitrary sequence without repetitions that contains every object $o \in dom(h)$ (except *null*). Let $\bar{v}[i]$ denote the object at position i in $\bar{v}$. For every instance variable x declared in some class E that is part of the scope of class D, let $\bar{v}(x_E)$ be the sequence of the same length as $\bar{v}$ such that the i-th element of $\bar{v}(x_E)$ is $h(\bar{v}[i])(E)(x)$, if $\bar{v}[i] \in dom(E)$, and $\mathsf{init}([x])$ otherwise. Let $\hat{\mathsf{f}} : dom(\mathsf{object}) \to dom(\mathsf{int})$ denote a function such that $\hat{\mathsf{f}}(null)$ denotes the length of $\bar{v}$, and for every object stored at some index i of $\bar{v}$ we have $\hat{\mathsf{f}}(\bar{v}[i]) = i$. Let $s^\dagger$ be the stack that is obtained from s by assigning $\bar{v}$ to $\mathcal{H}$, $\hat{\mathsf{f}}$ to f, and $\bar{v}(x_E)$ to $\mathcal{H}(x_E)$, for every sequence $\bar{v}(x_E)$. Finally, let $s^* = s^\dagger[\mathsf{result} \mapsto v]$.

Our first goal is to show that the antecedent of (8.4) holds in the state (s^*, h'). It is not difficult to see that $(s^*, h') \models \mathsf{heap}^D_V$. From $(s, h) \models (8.2)$ follows $(s^*, h) \models (8.2)$ by the construction of s^* (the relevant variables and function symbols do not occur in (8.2)). Then $(s^*, h') \models \mathsf{g}'(P') \wedge \bigwedge_{i \in I}(p_i \ \mathsf{instanceof}\ [p'_i])$ using Lemma 5.7.

Next, we prove that $(s^*, h') \models (\forall \bar{z} \in \mathcal{H} \bullet \lfloor P \rfloor \to (Q[\lfloor . \rfloor / \mathsf{old}(.)]))$. Let $\bar{\alpha}$ be an arbitrary sequence consisting of admissible values for the variables in $\bar{z}$ such that $(s^*[\bar{z} \mapsto \bar{\alpha}], h') \models \lfloor P \rfloor$. Because result does not occur in $\lfloor P \rfloor$ we also have $(s^\dagger[\bar{z} \mapsto \bar{\alpha}], h') \models \lfloor P \rfloor$. Note that $\bar{z}$ does not contain freeze variables. Using Lemma 5.7 we obtain $(s[\bar{z} \mapsto \bar{\alpha}], h) \models P$.

The existence of our initial computation $\langle S, (s, h)\rangle \to (s', h')$ implies that we also have $\langle S, (s[\bar{z} \mapsto \bar{\alpha}], h)\rangle \to (s'[\bar{z} \mapsto \bar{\alpha}], h')$ because the existence of a computation does not depend on the values of logical variables. Then $\models \{P\}\ m@C\ \{Q\}$ and our last two results (cf. Definition 3.3) together imply that

$$(s'[\bar{z} \mapsto \bar{\alpha}][\mathsf{result} \mapsto v], h')(s[\bar{z} \mapsto \bar{\alpha}], h) \models Q \ , \tag{8.6}$$

since $v = \mathcal{E}[\![e]\!](s', h') = \mathcal{E}[\![e]\!](s'[\bar{z} \mapsto \bar{\alpha}], h'))$. Next, we will argue that we can replace s' in (8.6) by $s^\dagger$. Recall that every free logical variable (except result) in Q occurs in $\bar{z}$. Moreover, Q contains no free local variables. Hence the value of Q does not depend on s'. By replacing s' by $s^\dagger$ we obtain the term $(s^\dagger[\bar{z} \mapsto \bar{\alpha}][\mathsf{result} \mapsto v], h')(s[\bar{z} \mapsto \bar{\alpha}], h) \models Q$, which is in turn equivalent to $(s^*[\bar{z} \mapsto \bar{\alpha}], h')(s[\bar{z} \mapsto \bar{\alpha}], h) \models Q$ due to the construction of s^*. The freeze stack s can be replaced by s^* because the values of logical variables in the freeze stack never play a role. Then Lemma 8.4 yields $(s^*[\bar{z} \mapsto \bar{\alpha}], h') \models Q[\lfloor . \rfloor / \mathsf{old}(.)]$, which shows that the entire antecedent of (8.4) holds in the given state.

From our previous result we conclude that the consequent of (8.4) also hold in the given state: $(s^*, h') \models Q'[\mathsf{g}'(.)/\mathsf{old}(.)]$. Recall that $\mathsf{g}'(e) = \lfloor e[\bar{p}/\bar{p}'] \rfloor$. From Lemma 8.4 follows that $\mathcal{N}[\![\mathsf{g}'(e)]\!](s^*, h') = \mathcal{E}[\![e[\bar{p}/\bar{p}']]\!](s^*, h)$ for every expression e. Moreover, $\mathcal{E}[\![e[\bar{p}/\bar{p}']]\!](s^*, h)$ is equivalent to $\mathcal{E}[\![e[\bar{p}/\bar{p}']]\!](s, h)$ because the value of a program expression e does not depend on the values of logical variables. But then we also have $(s^*, h')(s, h) \models Q'[\mathsf{old}(.[\bar{p}/\bar{p}_j])/\mathsf{old}(.)]$.

The freeze variables do not occur in Q'. For that reason we also have $(s[\mathsf{result} \mapsto v], h')(s, h) \models Q'[\mathsf{old}(.[\bar{p}/\bar{p}_j])/\mathsf{old}(.)]$. This latter result is almost equivalent to our initial goal (8.5). In fact, the following two observations imply that (8.5) follows from it. Firstly, observe that s' assigns the same values to logical variables as s because s' is part of the final state of a computation that started in the state (s, h), and the values of logical variables are not changed during a computation. And secondly, recall that a postcondition like Q' only has occurrences of local variables inside expressions of the form $\mathsf{old}(e)$. Hence its value only depends on the values of local variables in the *freeze* state. □

8.2 Modular Adaptation Rules

We have seen in the previous section what behavioral subtypes are, and how one can verify whether two classes are behavioral subtypes. We have also claimed that behavioral subtyping enables modular proof systems. This section provides the details of such a proof system.

We have written in the introduction of this chapter that a modular proof system is a proof system that does not require respecification or reverification of existing code when new classes are added. In other words, existing proof outlines must remain valid under class addition. A simple inspection of our basic verification strategy in Section 7.1.1 reveals that only the set of verification conditions of dynamically bound method calls may change under class addition. The verification conditions of other statements remain unchanged.

The set of verification conditions of a dynamically bound method call is extended by class addition if the new class contains an implementation that may also be bound to the call. For one verification condition is generated for each method implementation that may be bound to the call. Two new sets of verification conditions are described in this section which both overcome this problem. These sets correspond to two new adaptation rules. Our first rule is based on Leavens and Weihl's *supertype abstraction* principle [LW90, LW95]. The second rule refines the first rule to overcome an incompleteness issue.

8.2.1 A Modular Adaptation Rule Based on Supertype Abstraction

Recall from Section 5.2.3 that our adaptation rule for dynamically bound method calls has one verification condition for each implementation that may be bound to a particular call. Leavens and Weihl [LW90, LW95] have pointed out that it suffices to generate only one verification condition for each method call if a program only contains behavioral subtypes. The verification condition that is indispensable is the one that corresponds to the specification of the method of the static (or nominal [LW90]) type of the receiver. This simplification is essentially sound because behavioral subtyping forces methods to refine the behavior of methods which they override. For this reason all other implementations also

satisfy the specification of the implementation that corresponds to the static type of the receiver. Hence one can reason about subtypes using the specifications of supertypes. This technique is known as *supertype abstraction*.

Supertype abstraction leads to a modular adaptation rule. The outline of that rule is

$$\frac{\mathsf{prov}([e_0], m) = C \qquad \{P\}\ m@C\ \{Q\} \qquad (VC_{\text{sup}})}{\{Q'\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q''\}\ .} \tag{8.7}$$

Recall that $\mathsf{prov}(C, m)$ denotes the class in which the implementation of method m that is used by C-objects is declared. We assume that the call occurs in the code of class D. The verification condition (VC_{sup}) is

$$\begin{aligned} \lfloor (\textstyle\bigwedge_{i=0}^{n} \mathsf{defined}(e_i)) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V^D \wedge \lfloor Q' \rfloor \wedge \\ (\forall \bar{z} \in \mathcal{H} \bullet \mathsf{g}(P \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (Q[\mathsf{rec}/\mathsf{this}][\mathsf{g}(.)/\mathsf{old}(.)])) \\ \rightarrow Q''[\mathsf{result}/u]\ , \quad (VC_{\text{sup}}) \end{aligned}$$

which is almost syntactically equal to the verification condition $(VC5)$ that we proposed in Section 5.2.2 for reasoning about statically bound method calls. We have only replaced heap_V by heap_V^D (as defined in Section 8.1.3) to make sure that (VC_{sup}) is a modular formula. All its other elements are defined in the same way as in $(VC5)$. This reveals that supertype abstraction basically reduces reasoning about dynamically bound method calls to reasoning about statically bound method calls!

It is not difficult to explain why this rule leads to a modular proof system. A proof system is modular if every existing derivation remains valid under class addition. We can prove that our new logic is modular by induction on the length of a derivation. In particular, we can show that every derivation that ends with an application of rule (8.7) remains valid. This requires us to prove that the premises of (8.7) still hold.

Observe that the value of the first premise $\mathsf{prov}([e_0], m)$ cannot be altered by program extensions because adding new classes has no effect on the existing inheritance chain of class $[e_0]$. The second premise of (8.7) follows from the induction hypothesis if the other rules of the logic are also modular. Finally, we can prove that class addition does not affect the validity of (VC_{sup}). Essentially, this holds because every subclass relation that is obtained by adding a new class to a program is always an extension of the old subclass relation. Thus we can show that we may still apply (8.7). Hence this rule leads to a modular proof system.

8.2.2 Subtype Awareness

Unfortunately, rule (8.7) has a potential weakness. This weakness is caused by the fact that overriding methods may have stronger specifications than overridden methods. Rule (8.7) does not always allow us to reason about method calls

using these stronger specifications. Hence it is too weak to prove certain valid Hoare triples. This problem has been observed by Soundarajan and Fridella [SF99].

The weakness can be illustrated using the code of Example 8.2. Consider the Hoare triple

$$\{a \text{ instanceof } \mathit{CachedAccount} \wedge a.\mathit{bal} = 4\}\ S\ \{((\mathit{CachedAccount})a).\mathit{cache} = 4\}\ , \tag{8.8}$$

where S is the call $a.\mathit{setBalance}(5)$. We assume here that the static type of the local variable a is *Account*. Observe that the precondition of the call contains additional information regarding the dynamic type of the receiver because it states that a denotes an object of some subtype of *CachedAccount*. This information implies that the call will *not* be bound to the implementation of *setBalance* in the superclass *Account* but to the implementation in *CachedAccount* or to an implementation in some unknown subclass of the latter class. Hence the given Hoare triple holds.

But can we prove (8.8) using rule (8.7)? This is unfortunately not the case since the specification of *setBalance* in *Account* is too weak to prove the required postcondition. It does not say anything about its effect on the *cache* field of its receiver. We can only prove the required Hoare triple if we are able to reason about the call using the stronger specification of the implementation of *setBalance* in subclass *CachedAccount*.

We believe that situations like the one above (where the specification contains additional information regarding the dynamic type of the receiver) occur quite often in object-oriented programs. Classes frequently store objects of some particular type in polymorphic data structures such as lists and enumerations. The static type of the objects in such structures is usually the root type object. Thus information about the dynamic types of these objects is lost when they are stored in these data structures. However, this information is then usually added to a class invariant (see Chapter 9 for invariants). In this way, it is available whenever methods are called on the objects in these data structures.

How can we solve the indicated problem? Observe that supertype abstraction enables us to reason about a call without knowing the specifications of all implementations that may be bound to the call. It uses the specification of the implementation of the static type of the receiver and *ignores* all other specifications even when these specifications are visible in the context of the call. Clearly, we can improve the situation by devising a rule which takes all visible specifications into account. We will use the term 'subtype awareness' for this technique.

But when is a specification visible? We say that a specification is visible in some class C if the class D in which it is declared is part of the scope of C. A specification is visible in the context of a call if it is visible in the class in which the call occurs.

8.2.3 An Adaptation Rule Based on Subtype Awareness

How can we change rule (8.7) in such a way that it also considers other visible specifications? Do we have to prove one verification condition for every visible specification? Fortunately, this will not be necessary because we will continue to assume behavioral subtyping. We just have to weaken (VC_{sup}) in such a way that we *may* also prove a given call specification using the stronger specification of a visible subtype.

Suppose that we consider a call $u := e_0.m(e_1, \ldots, e_n)$ in some class C. Let $C_1 = \mathsf{prov}([e_0], m)$, and let $C_2, \ldots, C_k$ be an enumeration of all other classes D such that

i) $D \in \mathsf{scope}(C)$,
ii) $D \preceq [e_0]$, and
iii) D also contains an implementation of m.

The outline of our new adaptation rule based on subtype awareness is then as follows.

$$\frac{\{P_1\}\ m@C_1\ \{Q_1\}, \ldots, \{P_k\}\ m@C_k\ \{Q_k\} \qquad (VC_{\text{sub}})}{\{Q\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q'\}}\ . \tag{8.9}$$

So all visible method specifications must hold in order to prove the call specification, but we only have to prove one verification condition (VC_{sub}). Let $\bar{p}_j$ be the formal parameter sequence of the implementation of m in class C_j, for $j \in \{1 \ldots k\}$. Moreover, let I_j abbreviate the formula

$$(\forall \bar{z}_j \in \mathcal{H} \bullet \mathsf{g}_j(P_j \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (Q_j[\mathsf{rec}/\mathsf{this}][\mathsf{g}_j(.)/\mathsf{old}(.)]))\ .$$

Thus I_j is the part of the verification condition of an adaptation rule that corresponds to a particular method specification (instantiated with the elements of the specification in class C_j). The variable list $\bar{z}_j$ in I_j is a list of all the logical variables that occur free in either P_j or Q_j (without result). By g_j we denote the function that replaces every assertion Q and every expression e by $\lfloor Q[e_0, \bar{e}/\mathsf{this}, \bar{p}_j] \rfloor$ and $\lfloor e[e_0, \bar{e}/\mathsf{this}, \bar{p}_j] \rfloor$, respectively, where $\bar{e}$ abbreviates the list $e_1, \ldots, e_n$.

The verification condition (VC_{sub}) of our adaptation rule (8.9) is the formula

$$\lfloor (\bigwedge\nolimits_{i=0}^{n} \mathsf{defined}(e_i)) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V^C \wedge \lfloor Q \rfloor \rightarrow \bigvee\nolimits_{j=1}^{k} (\lfloor e_0 \rfloor\ \mathsf{instanceof}\ C_j \wedge (I_j \rightarrow Q'[\mathsf{result}/u]))\ . \quad (VC_{\text{sub}})$$

The disjunction in (VC_{sub}) shows that it is sufficient to prove the desired postcondition Q' using only one specification I_j provided that the precondition implies that the receiver is an object of a type which inherits that particular implementation. All behavioral subtypes of C_j have implementations that also satisfy I_j. This observation explains why the clause $\lfloor e_0 \rfloor\ \mathsf{instanceof}\ C_j$ suffices here.

The new adaptation rule (8.9) does not rely on unknown classes and therefore fits in a modular proof system. But is the rule also sound? The following theorem states that rule (8.9) is sound for programs that enforce behavioral subtyping.

Theorem 8.6. *Rule (8.9) is sound in the context of a program π in which every type is a behavioral subtype of all its supertypes.*

Proof. Let $S \equiv u := e_0.m(e_1, \ldots, e_n)$ be a method call in some class C. Moreover, let $C_1 = \mathsf{prov}([e_0], m)$, and let $C_2, \ldots, C_k$ be an enumeration of all other classes D such that (i) $D \in \mathsf{scope}(C)$, (ii) $D \preceq [e_0]$, and (iii) D also contains an implementation of m. Finally, let $\models (VC_{\text{sub}})$ and $\models \{P_i\}\ m@C_i\ \{Q_i\}$, for every $i \in \{1 \ldots k\}$. We must prove the validity of $\{Q\}\ S\ \{Q'\}$. For this purpose we consider a computation $\langle S, (s,h)\rangle \rightarrow (s',h')$ and an additional freeze state (s'',h'') such that $(s,h)(s'',h'') \models Q$. We will prove that $(s',h')(s'',h'') \models Q'$.

The given computation can only be derived using rule MC_1 of the operational semantics. Hence the following must hold.

$$\mathcal{E}[\![e_0]\!](s,h) = o = (D,i) \tag{8.10}$$

$$\mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \text{ for } i \in \{1 \ldots n\} \tag{8.11}$$

$$\mathsf{meth}(D,m) \equiv t\ m(p_1, \ldots, p_n)\{\ S'\ \mathsf{return}\ e\ \} \tag{8.12}$$

$$\langle S', (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h)\rangle \rightarrow (s^\circ, h') \tag{8.13}$$

$$\mathcal{E}[\![e]\!](s^\circ, h') = v \neq \bot \tag{8.14}$$

$$s' = s[u \mapsto v] \tag{8.15}$$

Our next step is to build a stack in which the variables of the heap model have the same values as the corresponding locations in the initial heap h. Let $\bar{v}$ be an arbitrary sequence without repetitions that contains every object $o \in dom(h)$ (except *null*). Let $\bar{v}[i]$ denote the object at position i in $\bar{v}$. For every instance variable x declared in some class $E \in \mathsf{scope}(C)$, let $\bar{v}(x_E)$ be the sequence of the same length as $\bar{v}$ such that the i-th element of $\bar{v}(x_E)$ is $h(\bar{v}[i])(E)(x)$, if $\bar{v}[i] \in dom(E)$, and $\mathsf{init}([x])$ otherwise. Let $\hat{\mathsf{f}} : dom(\mathsf{object}) \rightarrow dom(\mathsf{int})$ denote a function such that $\hat{\mathsf{f}}(null)$ denotes the length of $\bar{v}$, and for every object stored at some index i of $\bar{v}$ we have $\hat{\mathsf{f}}(\bar{v}[i]) = i$. Let $s^\dagger$ be the stack that is obtained from s by assigning $\bar{v}$ to $\mathcal{H}$, $\hat{\mathsf{f}}$ to f, and $\bar{v}(x_C)$ to $\mathcal{H}(x_C)$, for every field x declared in some class C. Finally, let $s^* = s^\dagger[\mathsf{result} \mapsto v]$.

Observe that $D \preceq [e_0]$ (Lemma 2.4 and Lemma 2.2). Let $\mathsf{prov}(D,m) = E$. Note that we do *not* necessarily have $E \in \{C_1, \ldots, C_k\}$ because it is possible that class $E \notin \mathsf{scope}(C)$. We do know, however, that $\mathsf{meth}(D,m)$ denotes the implementation of method m declared in class E.

Next, observe that $\models (VC_{\text{sub}})$ implies $(s^*,h')(s'',h'') \models (VC_{\text{sub}})$. We will prove that the antecedent of (VC_{sub}) also holds in that state to be able to conclude its consequent. That is, we will prove that

$$(s^*, h')(s'', h'') \models \lfloor (\bigwedge_{i=0}^{n} \mathsf{defined}(e_i)) \rfloor \wedge \lfloor \neg(e_0 = \mathsf{null}) \rfloor \wedge \mathsf{heap}_V^C \wedge \lfloor Q \rfloor \ . \tag{8.16}$$

In order to do so, we will first prove that for every formula Q^* in which the logical variables of the heap model and result do not occur we have that

$$(s, h)(s'', h'') \models Q^* \text{ if and only if } (s^*, h')(s'', h'') \models \lfloor Q^* \rfloor \ . \tag{8.17}$$

Note that $(s, h)(s'', h'') \models Q^*$ is equivalent to $(s^*, h)(s'', h'') \models Q^*$ by the construction of s^* if the logical variables of the heap model and result do not occur in Q^*. We get $(s^*, h) \models$ (5.27), $(s^*, h) \models$ (5.28), and $(s^*, h) \models$ (5.29) by the construction of s^*. Moreover, (8.13) and Lemma 5.6 imply that $dom(h) \subseteq dom(h')$. Then $(s^*, h)(s'', h'') \models Q^*$ is equivalent to $(s^*, h')(s'', h'') \models \lfloor Q^* \rfloor$ according to Lemma 5.7.

Let us now return to (8.16). From (8.10) and (8.11) above follows that $(s, h) \models \bigwedge_{i=0}^{n} \mathsf{defined}(e_i)$, and by (8.17) then $(s^*, h') \models \lfloor \bigwedge_{i=0}^{n} \mathsf{defined}(e_i) \rfloor$. Similarly, $(s, h) \models \lfloor \neg(e_0 = \mathsf{null}) \rfloor$ follows from (8.10) and (8.17). Moreover, it is not difficult to check that $(s^*, h') \models \mathsf{heap}_V^C$ holds too. We have $(s^*, h') \models \mathsf{heap}^C$ by the construction of s^* (heap^C only depends on variables of the heap model). The local variables of the caller and this refer to objects in $dom(h)$ because s is consistent with h. By the construction of s^* these values also occur in $\mathcal{H}$. We must also prove that $(s^*, h')(s'', h'') \models \lfloor Q \rfloor$. Note that we have $(s, h)(s'', h'') \models Q$. Hence $(s^*, h')(s'', h'') \models \lfloor Q \rfloor$ using (8.17). So (8.16) holds.

From (8.16) we conclude that the consequent of (VC_{sub}) also holds in the given state. That is, we have

$$(s^*, h')(s'', h'') \models \bigvee_{j=1}^{k} (\lfloor e_0 \rfloor \ \mathsf{instanceof}\ C_j \wedge (I_j \rightarrow Q'[\mathsf{result}/u])) \ ,$$

This latter result means that

$$(s^*, h')(s'', h'') \models \lfloor e_0 \rfloor \ \mathsf{instanceof}\ C_j \wedge (I_j \rightarrow Q'[\mathsf{result}/u]) \tag{8.18}$$

for some $j \in \{1 \ldots k\}$.

An important fact follows from the first conjunct of (8.18). Note that this formula is equivalent to $(s, h) \models e_0$ instanceof C_j (cf. (8.17)). But then it must be the case that $\mathcal{E}[\![e_0]\!](s, h) = o = (D, i) \in dom(C_j)$. This in turn implies that $D \preceq C_j$ (cf. Lemma 2.2). In other words, the receiver of the method call is an element of a (behavioral) subtype of C_j. This ensures that the implementation of method m in E satisfies the specification of method m in C_j as described in Def. 8.2.

Next, we focus on the second conjunct of (8.18). Recall that I_j abbreviates

$$(\forall \bar{z}_j \in \mathcal{H} \bullet \mathsf{g}_j(P_j \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (Q_j[\mathsf{rec}/\mathsf{this}][\mathsf{g}_j(.)/\mathsf{old}(.)])) \ ,$$

where g_j is the syntactical operation that takes a formula Q and returns the formula $\lfloor Q[e_0, \bar{e}/\mathsf{this}, \bar{p}_j] \rfloor$ with $\bar{e} = e_1 \ldots, e_n$ and $\bar{p}_j = p'_1, \ldots, p'_n$. Our next goal is to show that $(s^*, h') \models I_j$.

Let $\bar{z}_j = z_1, \ldots, z_l$, and let $\bar{\alpha} = \alpha_1, \ldots, \alpha_l$ be an arbitrary sequence of values such that $\alpha_i \in dom([z_i])$, for $i \in \{1 \ldots l\}$. Moreover, we assume that $\alpha_i \in dom(\bar{v})$ or $dom(\alpha_i) \subseteq dom(\bar{v})$ if α_i is an object or a sequence of objects, respectively. Finally, we assume that $(s^*[\bar{z}_j \mapsto \bar{\alpha}], h') \models \mathsf{g}_j(P_j \wedge \mathsf{rec} = \mathsf{this})$. By expanding g_j we get

$$(s^*[\bar{z}_j \mapsto \bar{\alpha}], h') \models \lfloor (P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}_j]) \rfloor \ . \tag{8.19}$$

From (8.19) and Lemma 5.7 follows that

$$(s^*[\bar{z}_j \mapsto \bar{\alpha}], h) \models (P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}_j]$$

because the variables of the heap model do not occur in $\bar{z}_j$. Moreover, result does also not occur in the formula $(P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}_j]$, and therefore

$$(s[\bar{z}_j \mapsto \bar{\alpha}], h) \models (P_j \wedge \mathsf{rec} = \mathsf{this})[e_0, \bar{e}/\mathsf{this}, \bar{p}_j] \ .$$

Then $(s[\bar{z}_j \mapsto \bar{\alpha}][\mathsf{this}, \bar{p}_j \mapsto o, \bar{v}], h) \models (P_j \wedge \mathsf{rec} = \mathsf{this})$, with $\bar{v} = v_1, \ldots, v_n$, follows from Lemma 5.8. The clause $\mathsf{rec} = \mathsf{this}$ in this formula implies that $s[\bar{z} \mapsto \bar{\alpha}](\mathsf{rec}) = o$.

From $(s[\bar{z}_j \mapsto \bar{\alpha}][\mathsf{this}, \bar{p}_j \mapsto o, \bar{v}], h) \models P_j$ above we get

$$s[\bar{z}_j \mapsto \bar{\alpha}][\mathsf{this}, \bar{p}_j \mapsto o, \bar{v}][\bar{p} \mapsto \bar{v}], h) \models P_j[\bar{p}/\bar{p}_j] \wedge \bigwedge_{i \in I} (p_i \ \mathsf{instanceof} \ [p'_i])$$

using Lemma 8.2, which is in turn equivalent to

$$s[\bar{z}_j \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \models P_j[\bar{p}/\bar{p}_j] \wedge \bigwedge_{i \in I} (p_i \ \mathsf{instanceof} \ [p'_i]) \tag{8.20}$$

because this formula does not depend on the values of parameters in $\bar{p}_j$ (except if these variables also occur in $\bar{p}$).

Because D is a behavioral subtype of C_j we know that the implementation of m in class E satisfies the specification with the precondition of (8.20) and the postcondition $Q_j[\mathsf{old}(.[\bar{p}/\bar{p}_j])/\mathsf{old}(.)]$ (cf. (8.3)). Moreover, observe that the computation in (8.13) entails that we also have

$$\langle S, (s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \rangle \rightarrow (s^\circ[\bar{z} \mapsto \bar{\alpha}], h') \tag{8.21}$$

because the existence of a computation does not depend on the values of logical variables. Furthermore $\mathcal{E}[\![e]\!](s^\circ[\bar{z} \mapsto \bar{\alpha}], h') = \mathcal{E}[\![e]\!](s^\circ, h') = v \neq \bot$. For the final state of the computation we then have

$$(s^\circ[\bar{z} \mapsto \bar{\alpha}][\mathsf{result} \mapsto v], h')(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \models Q_j[\mathsf{old}(.[\bar{p}/\bar{p}_j])/\mathsf{old}(.)] \ . \tag{8.22}$$

Lemma 8.3 implies that (8.22) is equivalent to

$$(s^\circ[\bar{z} \mapsto \bar{\alpha}][\mathsf{result} \mapsto v], h')(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p}_j \mapsto o, \bar{v}], h) \models Q_j \ . \tag{8.23}$$

From the computation in (8.21) follows that $s^\circ[\bar{z}_j \mapsto \bar{\alpha}](\mathsf{this}) = o$ because assignments to this are not allowed. Similarly, we have that $s^\circ[\bar{z} \mapsto \bar{\alpha}](\mathsf{rec}) = o$ follows from $s[\bar{z} \mapsto \bar{\alpha}](\mathsf{rec}) = o$. Then (8.23) and Lemma 5.10 imply

$$(s^\circ[\bar{z} \mapsto \bar{\alpha}][\mathsf{result} \mapsto v], h')(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p}_j \mapsto o, \bar{v}], h) \models Q_j[\mathsf{rec}/\mathsf{this}] \ . \tag{8.24}$$

The formula $Q_j[\mathsf{rec}/\mathsf{this}]$ contains no free occurrences of local variables, and every logical variable that occurs free in Q_j (except result) occurs in $\bar{z}$. Moreover, $s^*(\mathsf{result}) = v$. Therefore

$$(s^*[\bar{z} \mapsto \bar{\alpha}], h')(s[\bar{z} \mapsto \bar{\alpha}][\mathsf{this}, \bar{p} \mapsto o, \bar{v}], h) \models Q_j[\mathsf{rec}/\mathsf{this}]$$

holds too. We can also replace the freeze stack s in this formula by s^* because the only variables in Q_j whose values depend on the freeze state are this and $p'_1, \ldots, p'_n$. Subsequently, we can use Lemma 5.9 to obtain

$$(s^*[\bar{z} \mapsto \bar{\alpha}], h') \models Q_j[\mathsf{rec}/\mathsf{this}][\mathsf{g}_j(.)/\mathsf{old}(.)] \ . \tag{8.25}$$

Equation (8.25) completes our proof of $(s^*, h') \models I_j$. Combined with (8.18) this result implies that $(s^*, h')(s'', h'') \models Q'[\mathsf{result}/u]$. If we expand s^* we get $(s^\dagger[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'[\mathsf{result}/u]$. Then we know that

$$(s[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'[\mathsf{result}/u]$$

holds too because the dual heap variables do not occur in Q'. Moreover, $Q'[\mathsf{result}/u]$ does also not depend on the value of u. Hence we also have $(s'[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'[\mathsf{result}/u]$. Then $(s'[\mathsf{result} \mapsto v], h')(s'', h'') \models Q'$ follows from the definition of s' and Lemma 5.10. This shows that our main goal $(s', h')(s'', h'') \models Q'$ also holds because result does not occur in Q'. □

8.3 Behavioral Modular Completeness

Behavioral subtyping raises interesting questions regarding the completeness of program logics that are based on this principle. We have argued elsewhere [PdB05a] that program logics which are based on behavioral subtyping cannot be (relatively) complete. The problem with the standard completeness notion is that it puts no constraints on programs other than the ordinary typing rules. It requires that every valid Hoare triple regarding such programs is derivable in a particular program logic. Consequently, one must also be able to derive every valid property of programs in which the behavioral subtyping principle is violated. We believe that this requirement is too stringent [PdB05a].

Our response to this issue has been to propose two novel completeness notions [PdB05a]. We have dubbed the most interesting of the two notions *behavioral modular completeness*. It is in particular relevant for the modular rules in

this chapter because it is supposed to define a level of completeness that can be reached by modular program logics. We will define and motivate behavioral modular completeness in this section.

8.3.1 Motivation

We have given a completeness proof for our non-modular proof outline logic in Section 7.3. In this proof, we annotated methods with their Most General Method Specifications. However, such specifications can only be expressed for closed programs because the freeze formulas in these specifications explicitly mention all program fields (cf. Section 7.3.3), which is not feasible for open programs in which the fields of certain (future) classes may be unavailable. Consequently, we must accept weaker specifications in completeness proofs of modular logics.

What is needed for modular program logics is a completeness notion that takes all method specifications for granted. It should only check whether the rules of a logic allow us to prove every call specification that follows from these specifications. This is a truly weaker completeness property because a method call may also satisfy specifications which do not follow from a particular specification of the corresponding method. The required completeness notion should essentially handle a method as a black box of which only the behavior as described by a particular specification is known. A completeness notion that satisfies these requirements has been proposed by Zwiers. He called this property *modular completeness* [Zwi87]. The completeness property of modular object-oriented program logics that we propose is a variant of Zwiers' completeness notion.

One important addition in our completeness notion is that it only requires a program logic to be able to derive properties of programs in which the behavioral subtyping principle is not violated. That it, is only considers programs in which every subtype is behavioral subtype of all its supertypes. Program logics that are based on behavioral subtyping will not even be sound in other contexts.

Another addition is that behavioral modular completeness forces a program logic to consider all method specifications which are visible. In other words, it is based on subtype awareness. Therefore we do not expect that proof rules which use supertype abstraction will lead to a program logic that is behavioral modular complete.

8.3.2 Definition

We will first state the definition of behavioral modular completeness, and subsequently explain its parts.

Definition 8.4. *A formal proof system is behavioral modular complete if, for every Hoare triple* $\{Q\}\ S\ \{Q'\}$ *of a statement* S *in some class* C *of an annotated program* π *in which every subtype is a behavioral subtype of all its supertypes,*

we have that

$$\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_l\}\ m_l@C_l\ \{Q_l\} \models \{Q\}\ S\ \{Q'\} \tag{8.26}$$

implies

$$\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_l\}\ m_l@C_l\{Q_l\} \vdash \{Q\}\ S\ \{Q'\}\ , \tag{8.27}$$

where $\{P_1\}\ m_1@C_1\ \{Q_1\}$, ..., $\{P_l\}\ m_l@C_l\{Q_l\}$ *are all the (valid) method specifications that are visible in* C.

There are several elements of this definition that deserve some clarification. We start with (8.27), which says that we can derive $\{Q\}\ S\ \{Q'\}$ using the assumptions $\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_l\}\ m_l@C_l\{Q_l\}$. The meaning of (8.26) is more difficult to define. It roughly says that $\{Q\}\ S\ \{Q'\}$ is a valid Hoare triple if all Hoare triples $\{P_1\}m_1@C_1\{Q_1\}, \ldots, \{P_l\}m_l@C_l\{Q_l\}$ are also valid.

One can formalize this validity notion by defining an operational semantics relation with members of the form $\langle S, (s,h)\rangle \overset{\text{SPECS}}{\longrightarrow} (s', h')$. The SPECS parameter of this relation denotes a finite set of method specifications that contains the specifications $\{P_1\}\ m_1@C_1\ \{Q_1\}, \ldots, \{P_l\}\ m_l@C_l\ \{Q_l\}$. We will define this 'black box' semantics in such a way that the relation $\langle S, (s,h)\rangle \overset{\text{SPECS}}{\longrightarrow} (s', h')$ only holds if the method specifications in SPECS imply that a computation of S that starts in (s,h) may terminate in the state (s', h').

The new semantics is defined as usual by a set of axioms and rules. These axioms and rules are in most cases obtained by simply adding the SPECS parameter to the corresponding rule or axiom of our standard semantics for statements in Section 2.2.3. For example, the rule

$$\frac{\mathcal{E}[\![e]\!](s,h) \neq \bot}{\langle u := e\ , (s,h)\rangle \overset{\text{SPECS}}{\longrightarrow} (s[u \mapsto \mathcal{E}[\![e]\!](s,h)], h)} \qquad (LA_{\text{bb}})$$

describes the new semantics of local assignments. The real differences between our ordinary semantics and the new semantics become visible in the rules for method calls.

We will use the method specifications in SPECS to define the meaning of method calls. However, this raises the question *which* method specification we use to define the meaning of a call. For it is possible that SPECS contains several specifications of implementations which could in principle be bound to the call.

Consider, for example, a call $e_0.m(\bar{e})$. It may be the case that the execution of this call is bound to the implementation of the method m that belongs to class $[e_0]$. Note that SPECS always contains the specifications of the method implementations that corresponds to the static types of receivers of method calls in S. (cf. Lemma 8.1). However, SPECS may also contain specifications of implementations of m from subclasses of $[e_0]$. In this kind of situation we will always pick the *most specific* available method specification for the dynamic type of the receiver. We say that a method specification in class C is more specific

than a method specification in class D if $C \preceq D$. By $msms(m, C, \text{SPECS})$ we denote the most specific method specification $\{P\}\ m@D\ \{Q\}$ in SPECS such that $C \preceq D$.

The 'black box' rule for basic method calls basically says that every computation of a call is possible as long as the initial state and the final state of the computation satisfy the precondition and the postcondition of the most specific method implementation, respectively.

$$\frac{\begin{array}{c}\mathcal{E}[\![e_0]\!](s,h) = o = (C,i) \\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for } i \in \{1 \ldots n\} \\ msms(m, C, \text{SPECS}) = \{P\}\ m@D\ \{Q\} \\ (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h) \models P \\ dom(h) \subseteq dom(h') \\ (s,h')(s,h) \models Q\end{array}}{\langle e_0.m(e_1, \ldots, e_n), (s,h)\rangle \overset{\text{SPECS}}{\longrightarrow} (s,h')} \qquad (MC_{\text{bb}})$$

We assume here that $p_1, \ldots, p_n$ are the formal parameters of the implementation of m in class D. We require that $dom(h)$ is a subset of $dom(h')$ to make sure that (s,h') is a valid state; the semantics of the body of the implementation normally ensures that this condition holds. The first occurrence of s in the premise $(s,h')(s,h) \models Q$ may be somewhat surprising; however, it can be justified by pointing out that Q contains no free occurrences of local variables.

Similar changes must be made to the rule for object creation because that rule involves a call to a constructor method; we omit the details. Instead, we now turn to the new rule for calls to methods that return a value because that rule is slightly more complex than the previous one.

$$\frac{\begin{array}{c}\mathcal{E}[\![e_0]\!](s,h) = o = (C,i) \\ \mathcal{E}[\![e_i]\!](s,h) = v_i \neq \bot \quad \text{for } i \in \{1 \ldots n\} \\ msms(m, C, \text{SPECS}) = \{P\}\ m@D\ \{Q\} \\ (s[\mathsf{this}, p_1, \ldots, p_n \mapsto o, v_1, \ldots, v_n], h) \models P \\ dom(h) \subseteq dom(h') \\ v \in dom(t) \cap dom(h') \\ (s[\mathsf{result} \mapsto v], h')(s,h) \models Q\end{array}}{\langle u := e_0.m(e_1, \ldots, e_n), (s,h)\rangle \overset{\text{SPECS}}{\longrightarrow} (s[u \mapsto v], h')} \qquad (MC'_{\text{bb}})$$

The rule shows that an arbitrary value v that satisfies the postcondition is returned. We assume in (MC'_{bb}) that t is the return type of method m in class D. The condition $v \in dom(h')$ should be dropped if t is a primitive type.

With the new semantics we can give a formal meaning to (8.26).

Definition 8.5. *We have* $\text{SPECS} \models \{Q\}\ S\ \{Q'\}$ *if and only if for every state* (s,h) *and compatible freeze state* (s',h') *such that* $(s,h)(s',h') \models Q$*, and every state* (s'',h'') *such that* $\langle S, (s,h)\rangle \overset{\text{SPECS}}{\longrightarrow} (s'',h'')$*, we have* $(s'',h'')(s',h') \models Q'$*.*

Modular behavioral completeness is meant to be a weaker property than relative completeness. The two additional assumptions of relative completeness

regarding the expressiveness of the assertion language and the completeness of the underlying proof system for the assertion language (cf. Section 7.3) are therefore also permitted for our new property.

8.3.3 Evaluation

By defining the $\overset{\text{SPECS}}{\longrightarrow}$-relation in terms of the most specific available method specification we get a larger set of valid specifications $\{P\}\ S\ \{Q\}$, and consequently a stronger completeness criterion. This decision presumably implies that program logics that are based on supertype abstraction are *not* behavioral modular complete. For such logics always reason about method calls using the specification that corresponds to the static type of the receiver, which may not be the most specific available method specification.

On the other hand, we do believe that our modular adaptation rule (8.9), which is based on subtype awareness, may well be lead to a proof system that is behavioral modular complete. However, at this point this is merely a conjecture. A full proof of this conjecture would involve a substantial revision of our completeness proof in Section 7.3. We finish this section by formalizing the crucial step in such a completeness proof. It is our hope that this will encourage someone to undertake the task of completing the proof.

Conjecture 8.7. *Let* $\{Q\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q'\}$ *be an arbitrary Hoare triple regarding a method call in some class* C *of an annotated program* π *in which every subtype is a behavioral subtype of all its supertypes. Let* C_1 *be* $\mathsf{prov}([e_0], m)$*, and let* $C_2, \ldots, C_k$ *be an enumeration of all other classes* D *such that (i)* $D \in \mathsf{scope}(C)$*, (ii)* $D \preceq [e_0]$*, and (iii)* D *also contains an implementation of* m*. We assume that* $\{P_j\}\ m@C_j\ \{Q_j\}$ *is the* valid *specification of method* m *in class* C_j*, for* $j \in \{1 \ldots k\}$*. Let* SPECS *be such that* $\{P_j\}\ m@C_j\ \{Q_j\} \in \text{SPECS}$ *for every* $j \in \{1 \ldots k\}$*. Then*

$$\text{SPECS} \models \{Q\}\ u := e_0.m(e_1, \ldots, e_n)\ \{Q'\} \text{ implies } \models (VC_{\text{sub}})\ .$$

8.4 Advanced Specification Constructs

We finish this chapter with a section on advanced specification constructs. Thus far, we have only considered traditional (Hoare) specifications involving preconditions and postconditions. Several other specification constructs have been proposed in the literature. This section provides an overview of some of the most important specification constructs. Invariants, however, are discussed in more detail in the next chapter.

The specification constructs that we discuss below have not been chosen at random. Our selection is motivated by an incompleteness issue that reveals itself when one tries to prove a specification involving extended state in a modular way. We will only discuss constructs which are relevant to that particular phenomenon. Naturally, we start our discussion with a description of the problem.

```
class C imports object, D {
  D d ;

  requires true;
  ensures this.d = old(p);
  C(D p) { this.d := p }

  requires true;
  ensures this.d.x = old(this.d.x) + 1;
  void incD() { this.d.inc() }
    ...
}
```

```
class D imports object {
  int x ;

  requires true;
  ensures this.x = old(p);
  D(int p) { this.x := p }

  requires true;
  ensures this.x = old(this.x) + 1;
  void inc() { this.x := this.x + 1 }
}
```

Figure 8.1: The code of two example framework classes.

8.4.1 Class Scopes and Extended State

It is important to realize that object-oriented programs are almost never entirely written by one person. Even small programs often reuse and extend functionality that is provided by standard framework classes. Developers can extend framework classes by declaring new subclasses. These subclasses may also contain additional field declarations. Such fields are usually not visible in the superclass. This leads to an interesting reasoning challenge for modular program logics. We will illustrate this challenge using some small pieces of code.

Fig. 8.1 lists the code and annotation of two (framework) classes. The first class C holds a reference to an element of the second class D, which stores an integer value in its x field. The initial value of that field can be increased by calling the *inc* method. Class C has a corresponding method *incD* that calls the *inc* method on the object that it references. The specifications of the methods in the framework classes express these properties.

Now suppose that someone wants to extend class D in such a way that it becomes possible to identify whether an object still has its initial value. For this purpose, he (or she) defines a new subclass *ED* with one additional field *init*, which has value true until the first call to method *inc* is completed. The code and annotation of class *ED* is listed in Fig. 8.2.

Observe that class *ED* is declared as a subtype of class D. This is possible because the implementation of the *inc* method in *ED* satisfies the specification of the method in D which it overrides.

Now suppose that the developer of class *ED* writes another new class in which he wants to reuse the existing code of class C. For this purpose, the new class F references a C object. Moreover, assume that for some reason it is necessary that this object has the additional functionality that is provided by class *ED*. Therefore it is initialized with an instance of class *ED* instead of D in the code in Fig. 8.3.

Anyone who tries to construct a proof outline for the specification of the

```
class ED extends D refines D imports D {
  boolean init ;

  requires true;
  ensures this.x = old(p) ∧ this.init = true;
  ED(int p) { this.x := p ; this.init := true }

  requires true;
  ensures this.x = old(this.x) + 1 ∧ this.init = false;
  void inc() { this.x := this.x + 1 ; this.init := false }
}
```

Figure 8.2: The code of class ED.

```
class F imports object, ED, C {
  C c ;

  requires true;
  ensures this.c.d.x = 2 ∧ this.c.d instanceof ED ∧ ((ED)this.c.d).init = false;
  F() {
    D u ; u := new ED(1) ;
    C v ; v := new C(u) ;
    this.c := v ; this.c.incC() ;
  }
}
```

Figure 8.3: The code of class F.

constructor method of this class will discover at least two problems. The first problem is that the specifications of the methods that are called in the constructor method *only* describe their effects on the fields that are actually changed. But they do not say which fields are *not* modified during a call. For example, with the current specifications we are unable to prove that the execution of the constructor method of class C does not change the value of the x field of its parameter. This is not a major problem because the missing information can be added to the method specifications. Moreover, in Section 8.4.2 we will discuss modifies clauses, which solve this issue in an even more satisfactory way.

The second problem is much more difficult to solve. It manifests itself when we try to prove the last clause $((ED)\mathsf{this}.c.d).init = \mathsf{false}$ of the postcondition. We know that this clause holds because the call to method *inc* in the body of the method *incD* will be bound to the implementation in class *ED* due to dynamic binding. However, the specification of *incD* does not describe the modification of the *init* field. We cannot blame the specifier of class C for this problem. For we cannot expect that the specification of a class reflects the potential modification of fields which are *not visible* in that particular class. In

our example, field *init* is declared in class *ED*, which is not imported by class *C*.

One could criticize this example by pointing out that the validity of the postcondition of the constructor method depends on 'hidden' knowledge. In particular, it depends on our knowledge about the implementation of method *incD*. The fact that this method satisfies it specification by calling the *inc* method on the object referenced by its *c* field is not something that can be deduced from its specification. In other words, client *F* is demanding more than what is promised in the contract of *incD*.

We agree that it is not advisable to base expectations about method behavior on implementation details. On the other hand, we also believe that it may sometimes be necessary to be able to prove specifications like the postcondition in our example. So the only proper response to this problem seems to be to provide additional specification constructs which would enable us to write stronger contracts. In the following sections we will study several advanced specification constructs in order to determine their contribution to this issue.

8.4.2 Modifies Clauses and Data Groups

We have seen in the previous section that some of the specifications in our example are too weak because they only reflect the changes which a method performs. They do not say which fields remain unmodified. Ideally, these weaker specifications would be sufficient. Because otherwise we would have to describe the effect of a method on the fields of all objects in the program state, which would result in rather lengthy specifications. Moreover, only a limited set of fields is usually visible in the context of a particular method. What seems to be missing is a clause that says "... and nothing else changes". This issue is known as the 'frame problem' in procedure specifications [BMR95].

The standard solution to this problem is to explicit list the set of fields that a method is allowed to modify in a *modifies clause*. Such a clause would have the form

$$\mathsf{modifies}\ e_1.x_1, \ldots, e_n.x_n \ ,$$

where $x_1, \ldots, x_n$ are field names and $e_1, \ldots, e_n$ are ordinary program expressions. A method m with such a clause in its contract would only be allowed to modify the heap locations that are denoted by these expressions (in the initial state of its execution). Moreover, every method that m calls would also have to respect this part of m's contract.

Whether a method m satisfies its modifies clause can partly be checked by a simple inspection of its body and the contracts of the methods that m calls. For example, if a particular field x does not occur in a modifies clause, then we can simply check whether there is no assignment of the form $e.x := e'$ somewhere in m's body. Additionally, we have to ensure that none of the modifies clauses of methods that are called in m's body contains an expression of the form $e.x$.

We have to do a bit more work for a field x which is listed in the modifies clause. Assume, for example, that the modifies clause contains two expressions

of the form $e.x$: $e_1.x$ and $e_2.x$. Then we have to prove that m *only* modifies the x fields of the objects denoted by e_1 and e_2. We use two fresh logical variables z and z' to verify this. We can express the required property by adding the clause $\neg(z = e_1) \wedge \neg(z = e_2) \wedge z.x = z'$ to m's precondition, and the clause $z.x = z'$ to m's postcondition. Thus z denotes an arbitrary object other than the objects denoted by e_1 and e_2, and z' denotes the value of this object's x field. We can build a proof outline for this property by (implicitly) adding the clause $\neg(z = \mathsf{old}(e_1)) \wedge \neg(z = \mathsf{old}(e_2)) \wedge z.x = z'$ to every assertion in m's body. Subsequently, we can use our standard verification condition techniques to verify the modifies clause.

Another question is how the additional information contained in the modifies clause can be embedded in the verification conditions of method calls. Again we have to distinguish two situations. For a field x that occurs in the modifies clause of a method we can add the two clauses described in the previous paragraph to the precondition and the postcondition of the method while computing the verification condition for the method call (cf. Section 8.2.3). These clauses add the additional information to the method specification. For a field x which does not occur in the modifies clause we have two options. In principle, we could add clauses of the form $z.x = z'$ to both the precondition and the postcondition of the method specification. Thus the specification expresses that the x field of every object is not changed during the method execution. We should only do that for fields which are visible in the context of the call. Adding such clauses for other fields is useless because the specification of the call cannot depend on such fields.

There is, however, a superior solution which results in shorter and simpler verifications conditions. We have proposed this alternative solution elsewhere [PdB03a]. The idea is to modify the result of the $\lfloor . \rfloor$ operator for expressions of the form $q.x$ if x is a field which does not occur in the modifies clause of a method. Recall from Section 5.2.2 that $\lfloor . \rfloor$ replaces expressions that denote heap locations by expressions which denote the initial values of these expressions in the dual heap. The idea is that the dual heap stores the initial values of all heap locations. However, we know that fields which do *not* occur in a method's modifies clause are not modified by that method. Consequently, there is no difference between the initial and the final values of the corresponding heap locations. And therefore there is no need to replace these expressions. So the idea is to define $\lfloor q.x \rfloor = \lfloor q \rfloor.x$ for every field x which does not occur in the modifies clause. If we do not replace an expression $e.x$ in the precondition by the corresponding dual heap expression $\mathcal{H}(x_C)[\mathsf{f}(\lfloor e \rfloor)]$, then we can directly use information regarding $e.x$ in the precondition to prove properties regarding $e.x$ in the postcondition. This improvement renders superfluous the additional clauses $z.x = z'$.

We will illustrate both approaches in an example.

Example 8.3. *Suppose that we call a parameterless method m in some class C with an empty modifies clause. Hence m does no modify any variables. Let us*

assume that this method has precondition true *and postcondition* true*, which is a rather weak specification. But the modifies clause will nevertheless enables us to prove certain call specifications.*

Suppose that we want to prove that $\{\mathsf{this}.x = 5\}\ \mathsf{this}.m()\ \{\mathsf{this}.x = 5\}$*. Let D be the class in which this call occurs. We use verification condition (*VC_{sup}*) on page 166 to reason about this call. In the first approach we add the clause* $z.x = z'$ *to the precondition and the postcondition. This yields the verification condition*

$$\begin{aligned}&\mathsf{defined}(\mathsf{this}) \wedge \neg(\mathsf{this} = \mathsf{null}) \wedge \mathsf{heap}_V^D \wedge \mathcal{H}(x_C)[\mathsf{f}(\mathsf{this})] = 5\\ &\quad \wedge (\forall z, z' \in \mathcal{H} \bullet (\mathsf{true} \wedge \mathcal{H}(x_C)[\mathsf{f}(z)] = z' \wedge \mathsf{rec} = \mathsf{this}) \rightarrow (\mathsf{true} \wedge z.x = z'))\\ &\qquad\qquad \rightarrow \mathsf{this}.x = 5\ .\end{aligned}$$

Note that we did not expand the abbreviation heap_V^D*. This formula contains the clause* $\mathsf{this} \in \mathcal{H}$*. This clause is necessary here to ensure that the verification condition holds.*

The second approach yields a much shorter and simpler formula. We do not have to add clauses to the precondition and postcondition:

$$\begin{aligned}&\mathsf{defined}(\mathsf{this}) \wedge \neg(\mathsf{this} = \mathsf{null}) \wedge \mathsf{heap}_V^D \wedge \mathsf{this}.x = 5\\ &\qquad\qquad \wedge ((\mathsf{true} \wedge \mathsf{rec} = \mathsf{this}) \rightarrow \mathsf{true}) \rightarrow \mathsf{this}.x = 5\ .\end{aligned}$$

This verification condition clearly holds, because we can use the clause $\mathsf{this}.x = 5$ *in the precondition to prove* $\mathsf{this}.x = 5$ *in the postcondition.*

Data Groups

Behavioral subtyping requires a certain constraint on modifies clauses. For we cannot allow that a method in a subtype which overrides a method in a supertype behaves differently. The necessary constraint is that the locations in the modifies clause of the overriding method must be a subset of the locations in the modifies clause of the overridden method.

However, this simple restriction is too rigid. It forbids all modifications of fields that are declared in subclasses which are out of scope in the superclass. We can illustrate this claim using the code of the example in Section 8.4.1. First, consider method *inc* in class *D* in Fig. 8.1. This method only modifies field x, so it seems reasonable to give this method the clause $\mathsf{modifies}\ \mathsf{this}.x$. Next, consider method *inc* in class *ED* on page 178. This method modifies both field x and field *init*. However, according to the proposed constraint it is only allowed to modify x.

Clearly, we want to allow modifications of fields which are declared in subclasses. Leino has proposed *data groups* for this purpose [Lei98, LPHZ02]. A data group is essentially an underspecified set of fields. Data groups can be used in modifies clauses to indicate that every field that belongs to that group may

be modified. So if g is the name of a data group, then $e.g$ in a modifies clause means that the location $e.x$ may be modified if x is an element of the group g.

Data groups are open: fields of subclasses can be added to data groups declared in superclasses. This enables subclasses to extend the set of modifiable heap locations of an overriding method by adding new fields to data groups which occur in the modifies clause of the overridden method. This basic idea can be extended in several ways [Lei98, LPHZ02].

In our example, we could define a data group g in class D and give the *inc* method of that class the modifies clause modifies this.x, this.g. Subsequently, we can add a tag to the declaration of field *init* in class *ED* indicating that it belongs to group g. Finally, we can give the implementation of *inc* in this class the same modifies clause as the method which it overrides. This choice clearly satisfies the constraints on modifies clauses of overriding methods, and it permits the modification of *init*.

Modifies clauses and data groups are important advanced specification constructs. However, they do not help us to solve the more fundamental reasoning problem that we observed in the previous section. We could use them to specify that method *incD* in class C may modify *init*, but we cannot use them to specify precisely *how* it is modified. The former could be done by adding the clause modifies this.$c.g$ to the specification of this method. This would signal to class F that *init* is potentially modified. But it does *not* reveal that this field invariably gets the value false during the call.

8.4.3 Trace Specifications

Another radical extension of the expressiveness of method specifications results if we allow specifications to describe the *trace* of a method. A trace is a finite sequence of communication events (or method calls in the context of object-oriented programs). Traces are sometimes also called *communication histories* [dRdBH+01].

Recently, Soundarajan and Fridella have proposed techniques for reasoning incrementally about object-oriented systems [SF04]. In particular, their paper considers the problem how a subclass can prove a (potentially stronger) specification for a method which its inherits from its superclass without reexamining its code. This is an interesting problem because the inherited method may call other methods which are overridden in the subclass. Hence its behavior may be different when invoked on objects of the subclass. We will briefly describe how they use trace specifications to solve this problem.

Each element of a trace represents a call to a dynamically-bound method that occurs during the execution of a method. The name of the method, the initial and final state of the call, the parameter values, and the result value are recorded for every call. Every method describes its trace in an additional enrichment specification, which is only available for subclasses. A new rule enables subclasses to derive a stronger specification for methods that it inherits. This rule allows a subclass to assume that every call to an overridden method

in the trace satisfies the stronger specification of the overridden method.

Can traces help us to solve our remaining reasoning problem? We believe that they may provide a viable solution. However, stronger techniques than those described by Soundarajan and Fridella are required to fully handle our example in Section 8.4.1. One shortcoming of their work is that it only addresses the relation between a base class and its subclass. In our example, the method that needs to have a stronger specification is located in a separate (third) class. We would like to infer a stronger specification for method *incD* in class C; the specifications for the methods *inc* in class D and its subclass ED are strong enough.

It seems that we could solve the issue if method *incD* would provide an enrichment specification to class F which would reveal that it satisfies its specification by calling the *inc* method on its D-object. Then class F could deduce that this would result in a call to the implementation of *inc* in the subclass ED because it knows that this D-object is actually an instance of the subclass ED. However, this requires that the enrichment specification is not only available for subclasses but also for arbitrary clients like class F. Moreover, there are also several aliasing restrictions in the language considered by Sounderajan and Fridella which are not satisfied by most object-oriented languages. However, it may well be possible to realize the necessary extensions of their system.

Chapter 9

State Based Invariants and Object Allocation

Sharing of objects is often necessary to increase the speed and reduce the resource demands of programs. A system that allocates too many objects is prone to be slow. This phenomenon forces modules to share objects whenever possible.

Fortunately, many objects can safely be shared. This holds, for example, for simple immutable objects like strings and classes that encapsulate primitive data like integers or floating-point values. More complex examples include objects that represent key strokes, and borders of graphical user interface elements.

The resource demands of a program can be reduced by means of a mechanism that handles requests of client code for new objects by returning existing, shared objects whenever possible. Moreover, clients should be discouraged (or downright precluded) from allocating such objects directly. The flyweight pattern [GHJV94] supports object sharing by means of *factories* that maintain pools of shared objects. It is interesting to see that many variants of this pattern appear in version 1.4 and later versions of the Java API.

In this chapter, we formally analyze this type of object sharing by studying the invariants that describe such object pools. A common feature of these invariants is that they are falsifiable by the allocation of new objects. This is a disturbing observation because object allocation is possible in every context in which a constructor method of the class is visible. Therefore almost every code fragment may potentially falsify the invariant. We show how such invariants can nevertheless be maintained by means of *creation guards*. A creation guard for a particular class is a formula that should hold in each state in which a new object of that class is allocated.

Invariants are commonly used to describe properties of the encapsulated representation of a single object following an influential paper by Hoare [Hoa72]. Barnett and Naumann recently proposed a friendship system with update guards for maintaining invariants over shared state [BN04]. In this chapter, we show

how to extend their system with creation guards in order to control object allocation.

This chapter is organized as follows. Section 9.1 presents an example involving a border factory that enables clients to share border objects. In the following section we describe the state based invariant framework, along with some results concerning the relationship between object creation and invariants. Section 9.3 summarizes the friendship methodology on which we build. Section 9.4 introduces our creation guards, and the corresponding invariant methodology. It also provides a sketch of its soundness proof. The last section is devoted to related work and conclusions.

9.1 An Example: Sharing Borders

In this section, we describe an example factory class that enables clients to share bevel borders. Fig. 9.1 shows the two classes of the example. The example is derived from the corresponding classes in Java's `javax.swing` package. More complex examples are possible, but this example suffices to illustrate our approach. The example is written in a simple class-based object-oriented Java-like language that extends our language cOOre (cf. Chapter 2). For convenience, we will assume that the examples in this chapter behave according to the semantics of Java. However, this does not imply that the techniques that we will describe cannot be applied to other object-oriented languages like C#.

Our example uses static fields and methods which are not present in cOOre. Static class elements are associated with the class in which they are declared and not with its instances. Static fields are essentially global variables.

A bevel border can either be lowered or raised. The *type* field in class *BBorder* stores the type of the border. The factory class has two static fields (*LOWERED* and *RAISED*) that represent the two types.

Factory methods handle request for specific objects. They ensure that only one object is created for each value (or type). The *getBBorder* method in the *BorderFactory* class is an example of a factory method; it returns *BBorder* objects. The border factory class typically provides a factory method for each available border type.

To enable object sharing, references to the existing instances of a class should be maintained in a global object pool. In our example two static variables (*raised* and *lowered*) are used to build such a pool; more complex examples usually involve a hash table.

The constructor method of class *BBorder* is public. This is necessary because the factory method must have access to it. However, this also implies that client code is able to ignore the factory method by directly instantiating the class. That is, the implementation does not effectively impose object sharing on clients. This kind of situation is often accompanied by strong warnings in the class documentation not to exploit this leak. We will show how creation guards can repair this weak spot.

```
class BBorder {
  private boolean type ;

  public BBorder(boolean type) { this.type := type ; }

  // methods for drawing the border omitted
}
class BorderFactory {
  public static final boolean RAISED := true, LOWERED := false ;
  private static BBorder raised, lowered ;

  public static BBorder getBBorder(boolean type) {
    if (type = RAISED) {
      if (raised = null) { raised := new BBorder(RAISED) ; }
      return raised ;
    }
    else {
      if (lowered = null) { lowered := new BBorder(LOWERED) ; }
      return lowered ;
    }
  }

  // fields and factory methods for other border types omitted
}
```

Figure 9.1: A class that represents bevel borders, and a factory class that maintains a border pool.

In situations where a factory only controls one class, it is best to place the factory method in the same class; the problem can then be avoided by declaring the constructor to be private (see, e.g., class `java.util.Currency` in version 1.4 of the Java API). However, one still needs a way to check whether statements in that class do not inadvertently falsify invariants by allocating new objects. Our example addresses the more general situation where the factory method resides in a different class.

The code of the factory class facilitates (but does not impose) an invariant regarding *BBorder* objects: each object occurs in the object pool, and each bevel border has a unique type. That situation is described by the following invariant.

$$
\begin{array}{ll}
(\forall b : BBorder \bullet b = lowered \lor b = raised) & \\
\land\ lowered \neq \mathsf{null} \rightarrow lowered.type = LOWERED & (BorderFactory.Inv) \\
\land\ raised \neq \mathsf{null} \rightarrow raised.type = RAISED\ . &
\end{array}
$$

(Here, and throughout this chapter, we assume that quantification ranges over allocated non-null objects.) We assign this invariant to class *BorderFactory*, which makes it a static invariant. We will assume that static invariants are

invariants that belong to a class, and not to the instances of a particular class.

Non-static (object) invariants describe the representation of *instances* of a class, and are allowed to refer to their receiver by means of this. The first part of the above invariant could be rephrased as the object invariant

$$\mathsf{this} = \mathit{lowered} \vee \mathsf{this} = \mathit{raised} \ .$$

However, this object invariant has the flaw that it makes the instances responsible for assigning themselves to the proper location in the object pool. This is impossible in our example due to the visibility restrictions. We will therefore focus on static invariants in this chapter.

Invariants reveal important design choices regarding a class that may justify code optimizations. For example, if invariant (*BorderFactory.Inv*) holds, then the following efficient implementation of the *equals* method is sufficient to check if two instances of the class represent the same border type.

$$\mathsf{public\ boolean}\ \mathit{equals}(\mathit{Object}\ \mathit{obj})\ \{\ \ \mathsf{return\ this} = \mathit{obj}\ ;\ \ \}$$

9.2 State Based Invariants

Invariants are commonly expected to hold in all 'visible' states. This implies that the invariants must hold every time control leaves a method of a class [HK00, LBR04]. However, it is also possible to use a state based approach to invariants [BDF+04] in which the state signals which invariants hold. This approach has been developed in the context of the Spec# programming system [BLS05]. It handles callbacks [BDF+04] and inter-object relationships [LM04, BN04] more flexible while still preventing scenarios in which one wrongfully assumes that an invariant holds.

We will explain the state based approach in more detail below. But first we describe the syntax of invariants and the relationship between invariants and object allocation.

9.2.1 The Syntax of Invariants

We will assume that invariants are expressed in terms of the expressions of the underlying (Java-like) programming language. That is, they should be based on the following set of expressions:

$$\begin{array}{rcl} e \in \mathit{IExpr} & ::= & \mathsf{null} \mid e.x \mid C.f \mid (C)e \mid e\ \mathsf{instanceof}\ C \\ & \mid & e = e \mid \mathsf{op}(e_1, \ldots, e_n) \mid z \mid \mathsf{undefined} \mid \mathsf{defined}(e) \end{array}$$

We will assume that $C.f$ denotes the value of the static field f declared in class C. The meaning of every other expression is similar to its meaning in cOOre (see Chapter 2).

Invariants are simply formulas over the set of expressions that we have defined above:

$$I \in \mathit{Inv} \quad ::= \quad e \mid \neg I \mid I \wedge I \mid (\forall z : t \bullet I)$$

Invariants must satisfy one additional syntactical restriction: they are not allowed to have unbound occurrences of logical variables.

We will denote the value of an invariant expression by $\mathcal{N}[\![e]\!](s,h)$ along the lines of the formal semantics in Section 3.1.1. The missing case is as follows: $\mathcal{N}[\![C.f]\!](s,h) = h(C.f)$. Note that we assume that a heap h also assigns values to static fields. This requires a simple extension of the domains of heaps; we omit these details.

We will write $\mathcal{P}[\![I]\!](s,h)$ to denote the value of an invariant I in the state (s,h). The definition of $\mathcal{P}[\![I]\!](s,h)$ is equal to the definition of $\mathcal{P}[\![P]\!](s,h)$ in Section 3.1.1. However, the quantification domain of an invariant $(\forall z : C \bullet I)$ does not include the null reference.

9.2.2 Quantification and Object Allocation

The static invariant of class *BorderFactory* (*BorderFactory.Inv*) is an example of an invariant that is falsifiable by object creation. It is clear that this invariant is falsified by the allocation of a new instance of class *BBorder* whenever the static variables *raised* and *lowered* already reference existing objects.

We can use the weakest precondition calculus for object allocation in Chapter 6 to check whether invariants are falsifiable by object allocation. Recall that the operation $[\mathsf{new}(C)/u]$ computes the weakest precondition of the allocation of a new instance of class C, and its assignment to a fresh local variable u that temporarily stores a reference to the object. This operation does not model the execution of a constructor method. It merely models the effect of the heap extension that is caused by the allocation of a new object, which is the first effect of the execution of a statement $\mathsf{new}\ C(e_1,\ldots,e_n)$ in Java [GJSB00]. Whether the constructor method preserves the invariant should be checked in the constructor method itself.

The following theorem describes the correspondence between falsifiability by object creation of an invariant and the validity of a formula.

Theorem 9.1. *An invariant I is falsifiable by the allocation of a new instance of class C if $\not\models I \rightarrow (I[\mathsf{new}(C)/u])$.*

Proof. We prove the contrapositive of the theorem. Let $(s,h) \models I$. Recall from Section 6.1 that $(s[u \mapsto o], h \cdot [o \mapsto \mathit{init}(C)])$ is the state that we obtain if we allocate an object of class C and assign it to the local variable u in the state (s,h). We have that $(s[u \mapsto o], h \cdot [o \mapsto \mathit{init}(C)]) \models I$ if I is not falsifiable by the allocation of a new instance of class C. Then $(s,h) \models I[\mathsf{new}(C)/u]$ follows from Lemma 6.10. □

In most cases one can also directly deduce from the syntax of an invariant that it cannot be falsified by the allocation of a new instance. The following result states that only invariants that quantify over a domain that includes the new object can be falsified by its allocation.

Theorem 9.2. *If an invariant I has no subformulas of the form $(\forall z : C \bullet I')$ or $(\exists z : C \bullet I')$, for some superclass C of class D, then it cannot be falsified by the allocation of a new instance of class D.*

Proof. By structural induction on I. □

We assume here that each class is also a sub- or superclass of itself. The implication is not valid in the opposite direction.

9.2.3 Relating Invariants and States

We will briefly summarize the state based invariant methodology (a.k.a. the Boogie approach) [BDF+04]. It uses auxiliary fields to signal whether invariants hold. For example, we can introduce an auxiliary field *inv* to signal which invariants hold for a particular object. Its value is always a superclass of the dynamic type of an object. The state based invariant methodology ensures that if the value of the *inv* field of an object is class C, then the object satisfies every object invariants declared in a superclass of C (including the invariant in class C). The following system invariant (for each class C) formally describes the relation between this field and the object invariant Inv_C of class C.

$$(\forall o : C \bullet o.inv \preceq C \rightarrow (Inv_C[o/\mathsf{this}])) \tag{9.1}$$

Here, $\preceq$ denotes the reflexive and transitive subclass-relation, and $[o/\mathsf{this}]$ is the capture-avoiding substitution of this by o defined in Chapter 4.

The default value of field *inv* is the root class object, which implies that the object invariant of this class must hold for each fresh object. The value of the *inv* field is controlled by two special statements, pack and unpack, which are defined as follows for a class D with immediate superclass C.

$\mathsf{pack}\ e\ \mathsf{as}\ D \equiv$
  $\mathsf{assert}\ \mathsf{defined}(e) \wedge \neg(e = \mathsf{null}) \wedge e.inv = C \wedge (Inv_D[e/\mathsf{this}])$;
  $e.inv := D$;

$\mathsf{unpack}\ e\ \mathsf{from}\ D \equiv$
  $\mathsf{assert}\ \mathsf{defined}(e) \wedge \neg(e = \mathsf{null}) \wedge e.inv = D$;
  $e.inv := C$;

The formulas that follow the assert keyword should be seen as preconditions for these statements. The program logic in which they are used should guarantee that they hold whenever these statements are executed (e.g., by adding them to the weakest preconditions of these statements). A runtime assertion checker can simply check if the assertion holds upon reaching the statement.

These statements enable a discipline whereby each object is sufficiently unpacked before its fields may be modified. This can be achieved by placing the

additional precondition $e.inv \not\preceq C$ on all field assignments of the form $e.x := e'$, where C is the class in which field x is declared.

The above mentioned discipline suffices for object invariants that only depend on the fields of their receiver. Other invariants can be allowed by using ownership [CPN98] in order to extend the range of invariants to objects beyond of the original object, to owned objects and objects with the same owner [BDF+04, LM04].

The Boogie approach to invariants can also be used to handle static invariants. For this purpose, we assign to each class an auxiliary boolean field *stable* that indicates whether the class is stable, i.e., whether its static invariant holds. Assignments to static fields of a class will only be allowed if it is unpacked. A class is unpacked if its *stable* field has the value false. The following statements control this field.

$$\begin{array}{rcl} \mathsf{pack\_class}\ C & \equiv & \mathsf{assert}\ C.Inv\ ;\ C.stable := \mathsf{true}\ ; \\ \mathsf{unpack\_class}\ C & \equiv & C.stable := \mathsf{false}\ ; \end{array}$$

The purpose of these statements is to maintain the following system invariant, for each class C.

$$C.stable \rightarrow C.Inv \tag{9.2}$$

9.3 The Friendship System

Barnett and Naumann [BN04, NB04] extended the set of admissible invariants by allowing object invariants to depend on fields of shared objects. The classes of these objects are called *friend* classes. Their extension allows, for example, the invariant of our factory class to depend on instance fields of objects of (friend) class *BBorder*. Note that *BorderFactory.Inv* depends on field *type* declared in class *BBorder*. We will describe in this section how their proposal can be applied to static invariants.

Their *friendship system* uses update guards to describe permitted updates to shared fields. An update is permitted if it occurs in a state in which the guards of the field hold, or if the factory class is unpacked. We will use the following syntax to declare an update guard U in class C for an instance field x of class C.

$$\mathsf{static\ guard}\ U\ \mathsf{guards}\ x\ \mathsf{for}\ F\ ;$$

The keyword static indicates that the guard protects a static invariant; the guard U protects the static invariant of class F against updates of field x of C-objects that would falsify F's invariant. The guard itself should be a valid formula that does not mention fields that are invisible to clients of the class. It may additionally refer to two keywords: this denotes the object whose field is modified, and val denotes the value that is assigned to the field. A field may have several update guards.

A static invariant $F.Inv$ that depends on a field x of some other class C is only allowed if it is sufficiently protected by the update guard of that field. An update guard U for field x protects the static invariant of class F if

$$F.Inv \wedge U \rightarrow (F.Inv[\mathsf{val}/\mathsf{this}.x]) \tag{9.3}$$

holds. By $[\mathsf{val}/\mathsf{this}.x]$ we denote the weakest precondition operation of the assignment $\mathsf{this}.x := \mathsf{val}$ (see Chapter 4).

As mentioned above, updates to guarded fields are only allowed in states in which either the guards hold or in which the class of the invariant is unpacked. This is checked by giving each assignment of the form $e.x := e'$ the (additional) precondition $\neg F.stable \vee (U[e, e'/\mathsf{this}, \mathsf{val}])$. Here $[e, e'/\mathsf{this}, \mathsf{val}]$ is the simultaneous substitution of this by e, and val by e'. Thus we ensure that all updates to this field maintain (9.2).

Note that guards are always placed in the class in which the field is declared. This is necessary in a modular proof system. It ensures that a proof of correctness for a method cannot be falsified by adding additional classes that declare new guards for arbitrary fields of other classes.

We can also use this mechanism to allow static invariants that depend on static variables declared in other classes. A guard declaration

$$\mathsf{static\ guard}\ U\ \mathsf{guards}\ f\ \mathsf{for}\ F\ ;$$

protects the static invariant of class F against updates of static variable $C.f$ in states in which the guard does not hold. The keyword this does not make sense in update guards of static fields and is therefore not permitted.

One can check whether an update guard U for a static field $C.f$ protects the invariant by replacing $[\mathsf{val}/\mathsf{this}.x]$ in (9.3) by the weakest precondition operation $[\mathsf{val}/C.f]$ of the assignment $C.f := \mathsf{val}$. An assignment $C.f := e$ is only allowed in states in which $\neg F.stable \vee (U[e/\mathsf{val}])$ holds.

In the following section we introduce creation guards in order to enable static invariants that are falsifiable by object creation. The definition of the set of admissible static invariants will therefore also be deferred to that section.

9.4 Creation Guards

Theorem 9.2 states that only invariants that quantify over a domain that includes new objects can be falsified by object creation. It would therefore be safe to allow static invariants to quantify over the instances of the class in which they are declared if that class has only private constructors. Thus creation of instances of the class would be restricted to methods of the class, and it would suffice to check that the methods of the class ensure that the invariant holds in all visible states.

However, as we have argued in Section 9.1, it is often not the case that the factory methods are part of the same class. Moreover, we often find protected or even public constructors for shareable objects. We will use creation guards to

grant the class of the factory method the right to quantify over shared objects. A creation guard for some class C is a formula that should hold in each state in which a new object of class C is allocated.

Let F be the class that contains the factory method(s) for objects of class C. Class C can protect the static invariant of its factory class by declaring a creation guard. Such a declaration could have the following form.

$$\mathsf{static\ creation\ guard}\ G\ \mathsf{for}\ F\ ;$$

The creation guard G is an arbitrary formula over the part of the program state that is visible to clients; it should not reveal hidden implementation details of class C.

The most commonly used creation guard is false. This creation guard seems to prohibit creation of objects of class C, but that is not the case. The effect of such a creation guard is that objects of that class can only be created if the factory class is unpacked. That is, we require that $\neg F.stable \vee G$ holds prior to the execution of each statement that allocates a new object of class C.

The invariants that are enabled by a creation guard depend on the strength of the guard. The only invariants that are allowed are those that cannot be inadvertently falsified by the allocation of a new object as a consequence of the creation guard. A creation guard G protects the static invariant $F.Inv$ of a factory class F against allocation of instances of class C if

$$F.Inv \wedge G \rightarrow (F.Inv[\mathsf{new}(C)/u])$$

holds, where u is a fresh local variable. Only invariants that are protected by guards in the above sense will be allowed to quantify over shared objects.

9.4.1 Admissible Invariants

In this subsection, we briefly summarize the methodology that we have proposed thus far. In particular, we give a precise definition of the set of admissible invariants that is supported by the guards that have been introduced up to now. The definition avoids possible complications that may arise due to subclassing; we discuss subclassing in Sect. 9.4.3.

Definition 9.1 (admissible invariant)**.** *A static invariant $F.Inv$ is admissible if the following conditions are met:*

- *each static variable $C.f$ that occurs in $F.Inv$ is syntactically distinct from $C.stable$, and either belongs to class F ($C \equiv F$), or class C has a static update guard U for $C.f$ and class F such that $F.Inv \wedge U \rightarrow (F.Inv[\mathsf{val}/C.f])$;*

- *each subformula of the form $e.x$ concerns a field x that is syntactically distinct from inv, and the class C in which it has been declared has a static update guard U for x and class F such that $F.Inv \wedge U \rightarrow (F.Inv[\mathsf{val}/\mathsf{this}.x])$;*

- *each subformula of the form* $(\forall z : C \bullet I)$ *concerns a class* C *that has a creation guard* G *for class* F *such that* $F.Inv \wedge G \rightarrow (F.Inv[\mathsf{new}(C)/u])$*; moreover, class* C *is either final, or has at least one private constructor, and no public or protected constructors.*

Note that quantification over the values of a primitive type is never a problem. Such formulas are not falsifiable by object creation (as follows from Theorem 9.2). The last clause ensures that the class over which the invariant quantifies has no subclasses. Classes that have at least one private constructor, and no public or protected constructors, cannot have subclasses in Java. Thus the quantification domain of a formula in an invariant never includes instances of subclasses. This prevents invariants from depending on the creation of such objects. We will weaken this restriction in Sect. 9.4.3, where we address subclassing.

9.4.2 The Border Example Revisited

In this subsection we revisit the example described in Sect. 9.1. A proof outline of the example classes can be found in Fig. 9.2. It shows what annotation is needed to ensure that the required invariant is maintained, and how the invariant can be used to guarantee that methods behave according to their specification.

The static invariant of class *BorderFactory* is introduced by the keywords static and invariant on three succesive lines; the actual invariant is the conjunction of its three parts. The first part of the invariant is protected by a creation guard in class *BBorder*. However, the given invariant is only admissible if the class *BBorder* would have been declared to be final; we will explain in the following section why the invariant is also admissible without finalizing class *BBorder*.

Note that the references to the static variables in the invariant do not require update guards; a static invariant is always allowed to depend on static fields of the class in which it is declared. The following two parts are protected by the update guard for field *type* in the code of class *BBorder*.

The proof outline does not restrict the values of the static fields *RAISED* and *LOWERED* with e.g., the invariant $RAISED = \mathsf{true} \wedge LOWERED = \mathsf{false}$. Instead, we assume that the proof method replaces occurrences of these variables by their initializer expressions, which would make the above invariant trivially true. This preprocessing step corresponds to the way Java compilers handle final static variables with initializer expressions that are compile-time constants [GJSB00, § 12.4.1].

The constructor method of class *BBorder* is listed with its precondition (requires clause) and postcondition (ensures clause). It assigns to field *type*, and must therefore require that the factory class is unpacked due to the update guard of the field. Note that we assume that an occurrence of a parameter in a postcondition denotes its value in the initial state.

The *equals*-method of class *BBorder* depends on the static invariant of *BorderFactory* as signalled by its precondition. It uses the fact that the class is

```
class BBorder {
  private boolean type ;

  static creation guard false for BorderFactory ;
  static guard false guards type for BorderFactory ;

  requires ¬BorderFactory.stable ; ensures this.type = type ;
  public BBorder(boolean type) { this.type := type ; }

  requires BorderFactory.stable ;
  ensures result = (obj instanceof BBorder
                    && ((BBorder)obj).type = this.type) ;
  public boolean equals(object obj) { return this = obj ; } }

class BorderFactory {
  public static final boolean RAISED := true, LOWERED := false ;
  private static BBorder raised, lowered ;

  static invariant (∀b : BBorder • b = raised ∨ b = lowered) ;
  static invariant raised ≠ null → raised.type = RAISED ;
  static invariant lowered ≠ null → lowered.type = LOWERED ;

  requires stable ;
  ensures result.type = type ∧ stable ;
  public static BBorder getBBorder(boolean type) {
    if (type = RAISED) {
      if (raised = null) {
        unpack_class BorderFactory ;
        assert type = RAISED ∧ raised = null ∧ (∀c : BBorder • c = lowered)
          ∧¬stable ∧ (lowered ≠ null → lowered.type = LOWERED) ;
        raised := new BBorder(RAISED) ;
        pack_class BorderFactory ;
      }
      return raised ;
    }
    else {
      if (lowered = null) {
        unpack_class BorderFactory ;
        assert type=LOWERED ∧ lowered=null ∧ (∀c : BBorder • c=raised)
          ∧¬stable ∧ (raised ≠ null → raised.type = RAISED) ;
        lowered := new BBorder(LOWERED) ;
        pack_class BorderFactory ;
      }
      return lowered ;
    } } }
```

Figure 9.2: A proof outline of the shared borders example.

packed and the system invariant (9.2) to prove its postcondition, which would otherwise be too strong.

The factory method preserves the invariant of the class according to its specification. It temporarily unpacks the class if it has to allocate a new instance of the class. For clarity, we have inserted assert statements that describe what holds immediately after the class is unpacked. The invariant is restored in the factory method by assigning the fresh object to the proper static variable after completion of the constructor method.

9.4.3 Subclassing

The set of admissible invariants that we defined in Sect. 9.4.1 does not allow quantification over a range that includes instances of subclasses. This may seem a strong restriction, but it actually matches well with many variants of the flyweight pattern that we found in the Java API. These sharing facilities do not cater for subclasses because the creation statements in the factory methods fix the classes of the objects in the pool.

However, it is not difficult to conceive a more flexible factory based on the prototype pattern [GHJV94] that would not statically fix the class of its objects. Such a factory method would have to be initialized with a prototype object. From that point on, the factory method should clone the prototype object each time a new object is required, thus ensuring that all objects have the same type. We therefore investigate the use of creation guards in the presence of subclassing in this section.

Assume that we have a static invariant in class F that quantifies over the instances of class C. In closed programs, one can check for each subclass D of C if its creation guards protect the invariant. However, this solution cannot be applied if some of the (future) subclasses are unavailable. Therefore we will have to rely on a system in which the creation guards of a subclass are restricted by the creation guards of its superclass.

One may be tempted to believe that it suffices to let subclasses inherit the guards of their superclass, but that is not the case. Assume, for example, that we have a static invariant $(\forall o : C \bullet \neg(o \text{ instanceof } D))$, where D is a subclass of C. This invariant is not falsifiable by creation of instances of class C, so we could give class C the creation guard true for this invariant. However, we can easily break the invariant by allocating an instance of class D. The inherited creation guard does not prevent this scenario. The problem with this invariant is that it depends on a property of objects that is not inherited by subclasses. Instances of a subclass belong to a different class than instances of their superclass.

The instanceof operator and the cast operator are examples of operators that depend on the class of the objects to which they are applied. One of their operands is a class name. This operand always reveals the criterion that is used. Fortunately, these operators cannot discriminate between instances of that class and its subclasses. It suffices to ensure that the creation guard of the mentioned class protects the invariant. However, this latter restriction needs

only hold if the operator occurs in a formula that quantifies over the instances of some superclass of the mentioned class.

These additional restrictions suffice to protect invariants provided that creation guards are inherited by subclasses. A subclass may override a creation guard that it inherits if the new creation guard is stronger than the inherited guard. The above considerations lead to the following refinement of Def. 9.1.

Definition 9.2 (admissible invariant)**.** *A static invariant F.Inv is admissible if the first two conditions of Def. 9.1 are met, and moreover, each subformula of the form* $(\forall z : C \bullet I)$ *of F.Inv concerns a class C with a creation guard G for class F such that the implication* $F.Inv \wedge G \rightarrow (F.Inv[\mathsf{new}(C)/u])$ *holds, and*

- *class C is either final, or has at least one private constructor, and no public or protected constructors, or*
- *every subclass D of C that occurs in I has a creation guard* G' *for F s.t.* $F.Inv \wedge G' \rightarrow (F.Inv[\mathsf{new}(D)/u])$.

9.4.4 Soundness

Soundness of our methodology means that system invariant (9.2) holds in every reachable state of a properly annotated program in which all invariants are admissible according to Def. 9.2. We presuppose a sound proof system which ensures that the explicated preconditions of program statements hold. We show that these preconditions suffice to ensure that the various statements in the program maintain the system invariant.

A full soundness proof would duplicate many steps in the soundness proof of the friendship system [NB04]. We will therefore only prove the results that cover the part of the proof that checks whether object allocation preserves (9.2).

A statement $\mathsf{new}\ C(e_1, \ldots, e_n)$ first allocates a new instance of class C, and then initializes the object by calling the corresponding constructor method with parameters e_1 to e_n [GJSB00]. For simplicity, we assume that parameter evaluation has no side effects. The methodology must ensure that the system invariant is maintained by the allocation to prevent scenarios in which the specifier of the constructor method wrongfully assumes that the invariant holds.

The following definitions play an important role in the proof. Let $\mathsf{classes}(I)$ denote the least set such that $C \in \mathsf{classes}(I)$ whenever invariant I has a subformula of the form $(\exists z : C \bullet I')$ or $(\forall z : C \bullet I')$, or a subexpression of the form $(C)e$ or $e\ \mathsf{instanceof}\ C$. By the *most specific superclass* of a class D in an invariant I we mean the most specific superclass C of D such that $C \in \mathsf{classes}(I)$, if any. Formally, $C \in \mathsf{classes}(I)$ is the most specific superclass of D in an invariant I if $D \preceq C$, and there exists no other class $E \in \mathsf{classes}(I)$ such that $D \preceq E$ and $E \preceq C$.

The following lemma shows that the allocation of an instance of an arbitrary subclass has the same effect on an invariant as the allocation of an object of its most specific superclass.

Lemma 9.3. *Let I be an arbitrary invariant. Let class C be a superclass of class D such that C is the most specific superclass of D in I if it exists. Let (s, h) be an arbitrary state, and let o_c and o_d be arbitrary objects of class C and D, respectively, that do do not occur in $dom(h)$. Then*

$$(s[u \mapsto o_c], h \cdot [o_c \mapsto init(C)]) \models I \iff (s[u \mapsto o_d], h \cdot [o_d \mapsto init(D)]) \models I \ .$$

Proof. By structural induction on I. For the base case one must prove that

$$\mathcal{E}[\![e]\!](s[u \mapsto o_c], h \cdot [o_c \mapsto init(C)]) = \mathcal{E}[\![e]\!](s[u \mapsto o_d], h \cdot [o_d \mapsto init(D)])$$

for every invariant expression e. □

The main result of this section implies that any admissible invariant cannot be falsified by object allocation in a state in which all relevant creation guards hold. It is necessary to prove a slightly stronger result to be able to prove the claim by structural induction.

Lemma 9.4. *Let $C.Inv$ be an admissible invariant according to the additional requirements stated in Def. 9.2 (i.e., without the requirements of Def. 9.1 concerning update guards). Let D be an arbitrary class. Let (s, h) be a state such that $(s, h) \models C.Inv$, and moreover, for each creation guard G declared in some class $E \in \mathsf{classes}(C.Inv)$ that protects some friend class C we have that $(s, h) \models G$. Then $(s[u \mapsto o], h \cdot [o \mapsto init(D)]) \models C.Inv$, where o is an object of class D such that $o \notin dom(h)$.*

Proof. By structural induction on I. We first prove for the base case that, for every invariant expression e, $\mathcal{E}[\![e]\!](s[u \mapsto o], h \cdot [o \mapsto init(D)]) = \mathcal{E}[\![e]\!](s, h)$ by structural induction on e.

The most interesting case of the lemma concerns an invariant $C.Inv$ such that $C.Inv \equiv (\forall z : E \bullet I)$ for some class E such that $D \preceq E$. Let F be the most specific superclass of D in $C.Inv$. Note that F exists because E is already a valid candidate. We have by Def. 9.2 that $C.Inv \wedge G \rightarrow (C.Inv[\mathsf{new}(F)/u])$, where G is the creation guard declared in class F for friend class C. Let o_f be an object of class F such that $O_f \notin dom(h)$. From $(s, h) \models C.Inv[\mathsf{new}(F)/u]$ follows by Lemma 6.10 that $(s[u \mapsto o_f], h \cdot [o_f \mapsto init(F)]) \models C.Inv$. The required validity of $(s[u \mapsto o], h \cdot [o \mapsto init(D)]) \models C.Inv$ then follows from Lemma 9.3. □

9.5 Related Work and Conclusions

The problem of maintaining invariants that are falsifiable by object creation has not been solved before. This is somewhat surprising because it is quite common to allow quantification in program annotations, and quantification is the (potential) source of the issue. Leino and Nelson identified the problem [LN02], but they responded to it by forbidding this kind of invariant.

Calcagno et al. [COB03] studied the consequences of garbage collection (object deallocation) on program specifications. They rightly pointed out that

certain formulas that are similar to the set of invariants that we studied are vulnerable to object deallocation. Their remedy is to weaken the semantics of quantification such that non-existing objects are also included. However, the invariants that we studied are not valid in their semantics. Consequently, one has no means to prove the correctness of method specifications that rely on such invariants. It is more common to ignore garbage collection in the semantics of garbage-collected languages without pointer arithmetics such as Java and C#. Note that our example invariants are invulnerable to garbage collection because the references retained by each factory ensure that the objects are always reachable.

The Boogie approach to invariants was initially designed to handle reentrant calls to objects that are not in a stable state [BDF+04]. Several later extensions showed that the initial extended state approach could be stretched to cope with other object-oriented specification patterns. In this chapter, we have focussed on the use of creation guards, and have therefore ignored some of the orthogonal extensions such as the use of ownership.

Leino and Müller [LM04] proposed a new ownership model to support object invariants that depend on fields of owned objects that are not statically reachable from their owner, which allows, e.g., the invariant of a List object to depend on the fields of all the Node objects in its representation. They later also explored the use of ownership in static invariants [LM05]. However, quantification over owned objects is too weak to fully express the properties obtained by factory methods. They also explore quantification over packed objects, which turns out to be difficult to handle in a general way [LM05].

Barnett and Naumann [BN04] introduced update guards to protect object invariants that depend on fields of *friend* objects. They show how the set of friends can be managed using auxiliary state. Their friendship system protects invariants over shared state in circumstances where the ownership relation would be too rigid. They give a semantical characterization of the set of admissible invariants that rules out invariants that can be invalidated by object creation. An elaborate soundness proof of the system appeared in a companion paper [NB04].

The Java specification language JML [LBR04] defines static invariants in terms of visible states. Such a definition seems incompatible with invariants that are falsifiable by object creation because objects can be created in every state.

This chapter is a slightly revised version of an earlier paper [PCdB05]. A preliminary version of this work [PCdB04] has been presented at a workshop.

9.5.1 Conclusions

Object sharing is an important technique to overcome some of the potential resource demands and speed limitations of object-oriented programs. This is witnessed by the amount of examples of patterns that manage object allocation that we found in the Java API. However, as we have shown in this chapter,

the invariants that describe pools of shared objects are falsifiable by object allocation. The singleton pattern [GHJV94] is another example of a pattern that leads to an invariant that is falsifiable by object creation [PCdB04].

The main contribution of this chapter is a sound and modular methodology for static invariants which could be falsified by both states updates and object allocation. We introduced creation guards to maintain such invariants. The examples that we studied are best described using static invariants, but creation guards can also be used to protect object invariants.

The invariant methodology can be applied in both a static-checking and in a full program verification context. It also seems useful to check creation guards using a runtime assertion checker. The methodology is not tied to a specific program logic, although we have partly expressed it in terms of our previous work on program logics for object-oriented programs. The formulas that use the weakest precondition operation for object allocation can be rephrased in terms of the semantics of object allocation. We employed a syntactical description to be more specific about the set of admissible invariants.

We have implemented the invariant methodology in a tool that computes the proof obligations of proof outlines of sequential Java programs. This tool is described in the next chapter of this thesis.

Chapter 10

Tool Support

The preceding chapters of this thesis contain theoretical work. In this penultimate chapter we will show the more practical side of our research. We will describe a tool that supports our proof outline logic. This tool gives an impression of how the logic could be applied in practice.

We also hope that this chapter provides more insight into the design decisions behind our logic. We have already motivated the design of our proof outline logic in the introduction of this thesis. But the architecture of our verification tool also reflects some of the constraints that our logic satisfies. In particular, it shows that a proof outline logic enables us to automate important parts of the verification process.

The Verification Front-End Tool (VFT) that is described in this chapter has been developed by the author of this thesis. It is written in Java. The VFT is the successor of another tool that computes the verification conditions of annotated object-oriented flowcharts [PdB03b].

In the next section we explain the functionality of the VFT. The details of its architecture can be found in Section 10.2. We report our experiences with the tool in Section 10.3. The last section of this chapter contains an overview of related tools.

10.1 Using the VFT

The VFT is a stand-alone application that looks like a standard integrated development environment. It has a graphical user interface with several windows in which the user can manage and edit the files that contain the source code of a particular program. Syntax highlighting makes the structure of the code easier to detect, and drop-down menus and toolbar buttons provide access to the tool's functionality. Figure 10.1 shows the graphical user interface of the tool.

The differences between the VFT and a normal integrated development en-

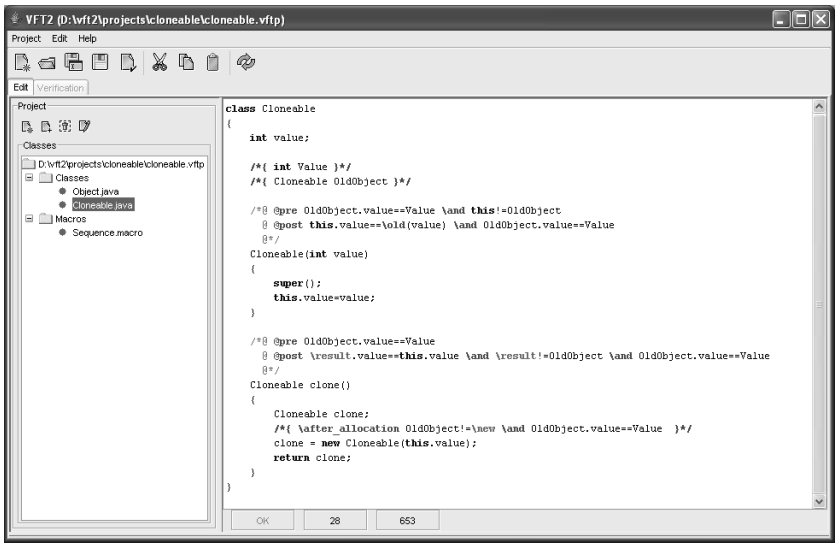


Figure 10.1: The graphical user interface of the VFT.

vironment start to become clear when the developer adds annotation to the Java source code. The VFT recognizes code annotation and highlights it accordingly. It expects the user to supply a precondition and a postcondition for every method. It also supports intermediate assertions in method bodies and logical variable declarations.

The assertion language which the VFT supports is similar to the cOOral language in Chapter 3. However, the precise syntax of the language differs somewhat. More information about the syntax of VFT's assertion language can be found in the documentation that is available at its website:

`http://www.cs.uu.nl/groups/IS/vft/`.

All annotation elements must be written between the standard Java comments markers `/*` and `*/`. This ensures that the annotation is ignored by Java compilers. Thus every proof outline is also a valid Java program. A user can easily insert these markers by selecting them in a popup menu that appears when he clicks on his right mouse button.

Another useful annotation feature of the tool is its support for *assertion macros*. A macro is an abbreviation for some specific (parameterized) formula. Macros can be used in proof outlines to shorten specifications and to improve their readability. Macros are defined in separate files. Their declarations are very similar to method declarations. A macro definition consists of a name, a formal parameter list, and an formula (the body of the macro).

Annotated programs can be compiled. This process consists of two stages. First, the VFT type checks both the program and its annotation. It also checks

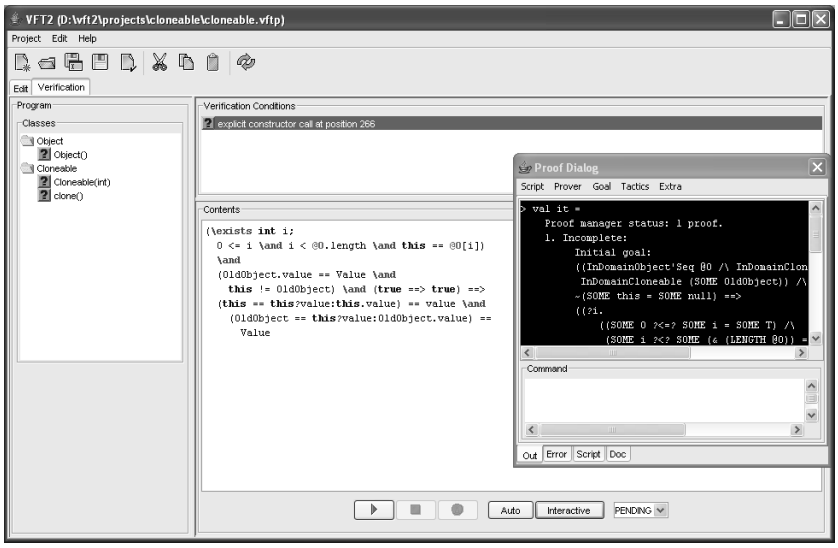


Figure 10.2: The verification panel and a proof dialog.

whether the program satisfies the normal sanity constraints for Java programs [GJSB00]. Violations of the typing rules and other constraints are reported in a separate window. The tool also produces a warning if the code contains Java constructs that are currently not supported. The Java subset which the VFT currently supports is similar to the cOOre language in Chapter 2. Additionally, it has some support for static fields and methods. And it also supports Java's increment and decrement statements [GJSB00].

During the second compilation stage the verification conditions of the proof outline are computed. This step always succeeds if the proof outline has passed the first compilation stage. Each method has its own set of verification conditions. The verification conditions are presented to the user on a separate tab panel (the verification panel) that is only accessible when the program has been compiled successfully. Figure 10.2 shows this panel (and a proof dialog which we will explain below).

The verification panel shows a tree structure which contains a node for every class and every method in the program. The verification conditions of a method are listed on the verification panel when its node is selected. For each verification condition a short description of its origin is shown. The actual verification condition is displayed when its description is selected.

The remaining task is to make sure that all verification conditions are valid. This is where the back-end of the tool comes into play. The VFT currently uses the HOL 4 prover[1] to verify its proof obligations. This automated proof system

[1]The HOL 4 system is available from `http://hol.sourceforge.net/`.

runs in the background during the execution of the tool. The VFT can send verification conditions to this system. After sending it reads the output of the prover to determine whether the proof obligation was proved. It then changes the status of the verification condition accordingly. The status of a verification condition is either *pending*, *valid*, or *invalid*. A small icon next to the description of a verification condition indicates its current status.

The easiest way to determine the validity of the verification conditions is to ask the VFT to try to prove them all automatically. The tool then sends all verification conditions to the theorem prover in a particular order. The prover gets a fixed amount of time to prove each verification condition. The tool resubmits the verification conditions that were not successfully proved in the first round when every verification condition has been examined. In a new round the prover either gets a longer time period for each verification condition or it is asked to try a different proof tactic. This process terminates when all proof obligations are proved or when it is interrupted by the user.

It is also possible to use HOL 4 interactively. The VFT supports interactive proof construction by means of a proof dialog that displays the output of the theorem prover. Figure 10.2 shows such a dialog. The user can send commands to the prover in a proof dialog by typing them in a particular field. It is also possible to select commands in the dialog's dropdown menu. The interactive mode can be used to prove the validity of complex verification conditions that cannot be proved automatically.

Obviously, it is also possible that the prover fails to prove a verification condition because it is invalid. Unfortunately, the tool currently does not detect invalid verification conditions. This is caused by the fact that HOL 4 does not systematically try to disprove formulas.

10.2 Architecture

The core activity of the VFT is data processing. Its input are annotated Java programs and its final output is a set of verification conditions. The data flow in the VFT is depicted in Figure 10.3.

The first step in the data processing chain is performed by a lexer and a parser. We have used the CUP[2] parser generator to build the parser. The parser cooperates with a lexer that was generated using the lexical analyzer generator JLex[3]. These tools produce a Java object structure that represents the abstract syntax tree of an annotated program.

In the next step, the VFT calls a method on the root of the object structure to initiate type checking. This method ensures that every variable in the abstract syntax tree is annotated with its proper type. But it also verifies that the program is well-defined. It checks, for example, whether each class in the program has a unique identifier.

[2]See `http://www2.cs.tum.edu/projects/cup/`.

[3]See `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

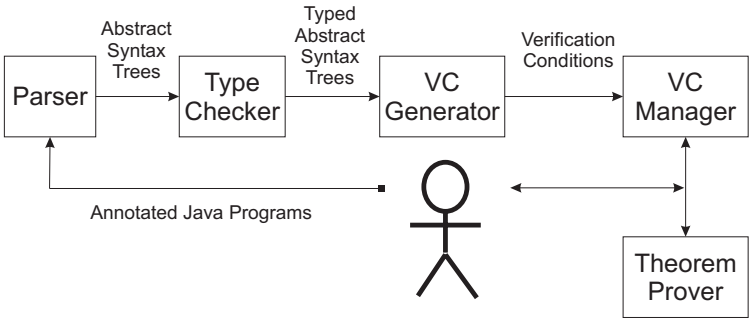


Figure 10.3: Data flow in the VFT.

The resulting typed abstract syntax tree is passed to the Verification Condition Manager. This manager can query the tree to obtain the verification conditions of a particular method. It then display these verification conditions to the user. But it can also obtain all verification conditions of the entire program and send them to the theorem prover. Each verification condition consists of a typed abstract syntax tree that represents a particular formula, a string that describes its origin, and an integer value that represents its status.

The final data processing step occurs when the Verification Condition Manager sends a verification condition to the theorem prover. It then constructs a string that represents the verification condition in the input language of the theorem prover. This is necessary because the HOL 4 system supports a higher order logic that differs from our assertion language. However, it is not difficult to represent formulas of our assertion language in this logic.

We will not provide a detailed description of the embedding of assertions in HOL 4's higher order logic because its details are fairly specific for this particular system. The translation simply follows the definition of the semantics of formulas in Section 3.1.1. This semantics sometimes refers to the subtype relation of the program. This relation is therefore also represented in the logic of the theorem prover. It is available as a program-specific *background axiom* for all verification conditions.

The current situation in which the VFT supports only one theorem proving system is not ideal. We would prefer a situation in which the user could select its preferred theorem prover from a range of systems. The architecture of the tool makes this feasible. For the background axiom and the resulting verification conditions are the only things that must be embedded in the theorem prover's logic. Moreover, the size of the assertion language is reasonably small. Its translation can therefore easily be adapted towards other theorem provers.

10.3 Evaluation

So far, the VFT has mainly been used for educational purposes. Students have worked with the tool during two editions of a course on program correctness at Utrecht University. The students wrote proof outlines and manually computed the corresponding verification conditions. Afterwards they compared their results with the verification conditions which the tool computed for the same proof outlines. They were also asked to prove some of the verification conditions in order to validate their proof outlines. This does not require much knowledge of the HOL 4 system because the verification conditions of small examples can usually be proved either automatically or by means of one of the standard proof procedures in the dropdown menu of the proof dialog.

The students proved that it only takes a few hours to learn to work with the VFT if one is familiar with programming in Java. Experienced Java developers can immediately start to add annotation to their code. But learning to write proper specifications takes more time of course.

One way to guide students towards writing useful specifications would be to add an extended static checking mode to the tool. This mode would force users to add at least enough annotation to ensure that methods do not result in, for example, null pointer exceptions. The VFT currently does not compel users to prove such properties. However, it is not difficult to adapt our proof outline logic accordingly. For example, if we want to make certain that an assignment $u := e$ terminates without exceptions then we must ensure that the value of e is defined. Recall from Chapter 4 that the weakest precondition of this statement with respect to a postcondition Q is the formula $\mathsf{defined}(e) \to Q[e/u]$. Changing this formula to $\mathsf{defined}(e) \wedge Q[e/u]$ additionally guarantees that the statement terminates without exceptions.

10.4 Related Work

Many verification systems have been built in recent years. Below, we will only discuss systems which can be used to verify object-oriented programs.

The VFT is similar to an extended static checker in the sense that it also automatically computes the verification conditions of proof outlines. A difference between the VFT and extended static checkers is that the VFT supports interactive theorem proving, whereas an extended static checker purely relies on an automated theorem prover. Another difference is that extended static checkers are designed to prove lightweight properties like the absence of runtime exceptions, whereas our tool currently does not direct the user towards proving some particular set of properties.

The first extended static checker for Java was ESC/Java [FLL+02]. Its successor ESC/Java2 [CK05] supports the JML [LBR04, LCC+02] specification language. It implements JML's visible state semantics for invariants.

The Spec# programming language [BLS05] extends C# with method spec-

ifications, invariants, non-null types and checked exceptions. There is a static programmer checker for Spec# (codenamed Boogie) that checks these properties. It is available as a plug-in for Microsoft Visual Studio. The Spec# programming language supports the state based invariant methodology [BDF+04] which we have discussed in Chapter 9.

Verger [Á05] is a command line tool that generates verification conditions for annotated Java programs in the input language of the PVS system. Its distinguishing feature is that it is able to handle multi-threaded programs. However, it neither supports inheritance nor subtype polymorphism.

Several tools generate verification conditions for annotated Java programs using existing verification tools for less complex languages. They translate Java programs and their specifications into these primitive languages and subsequently compute their proof obligations. This usually makes these proof obligations harder to understand. Moreover, the initial translation step can easily result in a logic that is incomplete or even unsound.

The Java Applet Correctness Kit (JACK) [BRL03] automatically computes verification conditions for Java Card applets with JML annotations. It does so by first translating Java classes into B models. The resulting proof obligations are B lemmas (instead of JML formulas). The Krakatoa tool [MPMU04] translates Java programs with JML specifications into the input language of the Why tool, which produces proof obligations for programs that are written in a ML-like minimal language.

Other tools support interactive verification using some specific logic. The KeY tool [ABB+05], for example, enables its user to interactively construct proofs in a dynamic logic for Java implementations of UML models with OCL [WK98] constraints. It is integrated in a commercial CASE tool (Borland's Together Control Center). Stenzel has developed a similar dynamic logic calculus for Java Card programs that is supported by the interactive KIV system [Ste04].

The Java Interactive Verification Environment (Jive) [MPH00] provides support for Poetzsch-Heffter and Müller's logic for Java programs [PHM99]. This logic is not suitable for verification condition generation. Users of Jive must therefore not only interact with a theorem prover component, but also with a program prover component. The latter component interactively constructs proofs using the rules of the aforementioned logic.

All tools that we have discussed sofar are based on logics that abstract from a particular formal semantics of object-oriented programs.The developers of the LOOP compiler [vdBJ01] follow a different approach. For this compiler translates Java programs with JML annotations into the higher order logic of PVS or Isabelle/HOL [Hui01] using an embedding of the (denotational) *semantics* of Java programs.

The LOOP compiler covers a significant subset of sequential Java. Moreover, it also properly models the semantics of Java's integral types [Jac03]. However, this approach also has some drawbacks. It is based on a complex encoding of Java's semantics in the higher order logic of the theorem prover. Such an encoding is difficult to learn. Most other tools only need an embedding of the

simpler and more compact semantics of specification formulas. Secondly, the proof obligations which the LOOP compiler constructs are larger than those of other logics. This is a consequence of the fact that all the reasoning takes place in the theorem prover. By contrast, our VFT already executes many reasoning steps when it generates the verification conditions of a program. The validation techniques which it implements encode many non-trivial reasoning steps.

Chapter 11

Conclusions

Our aim was to construct a proof outline logic for a language that has the object-oriented features of popular object-oriented languages like Java and C#. To this end, we have developed techniques for reasoning about assignments, method calls and creation statements which are valid in the context of a language that supports inheritance, subtype polymorphism and dynamic binding. Moreover, we have shown how these techniques can be used to validate proof outlines of object-oriented programs. The resulting logic is both sound and relatively complete for closed programs.

During our research we discovered that we could not build the desired proof outline logic from techniques that all reason in the same direction; our final logic therefore combines techniques that reason in the forward direction using strongest postconditions with techniques that reason in the backward direction using weakest preconditions.

The problems that we encountered are all related to dynamic object allocation. We found out that it is impossible to reason in the backward direction over method calls in a language with dynamic object allocation (cf. Section 5.2.2). For this would force us to reason in the initial state of a method execution about objects from its final state. This is not possible because the objects that are created during the execution of a method do not yet exist in its initial state. And our assertions can only describe objects that exist in the current state. We do not have this problem for the opposite direction because all objects from the initial state also exist in the final state.

The fact that our assertions can only describe existing objects is a consequence of the abstraction level of our logic. It is in principle possible to overcome this limitation, but only by creating a gap between the abstraction level of the programming language and that of its specifications. For normal program expressions in Java or C# only denote existing objects. Reasoning about future objects seems to require state parameters in the assertion language. Moreover, we would have to relate all expressions in assertions to some specific state.

It is also not possible to reason about all statements in the forward direction. We encountered difficulties while trying to define the strongest postconditions of object allocations. The proposed strongest postconditions are too weak to

prove certain postconditions which contain logical variables that do not occur in the corresponding preconditions (cf. Section 6.1.2). Our weakest precondition calculus for object allocation does not have this shortcoming. Hence our proof outline logic is a combination of techniques for both directions.

We have experimented in this thesis with expressions of the form $\mathsf{old}(e)$ in specifications. An expression $\mathsf{old}(e)$ denotes the value of the program expression e in the initial state of the current method execution. Thus such expressions can be used to refer to the initial values of expressions in, for example, the postcondition of a method. They seem to serve the same purpose as the logical variables in traditional Hoare logics [Apt81], which raises the question whether they render logical variables superfluous.

We have been able to integrate these expressions in our proof outline logic. Moreover, we have observed that their use simplifies the verification conditions of method calls (see our remark on page 81). But we have come to the conclusion that these expressions cannot totally replace logical variables in program logics for object-oriented languages. For the usage of logical variables that range over *finite sequences* remains necessary.

We have employed logical variables that range over sequences in the verification conditions of our adaptation rules (cf. Chapter 5) and in the freeze formula of our completeness proof (Chapter 7). In both cases we could not achieve the same goals with expressions of the form $\mathsf{old}(e)$. With logical variables that range over finite sequences we can model the dynamically allocated part of an object-oriented state, independent of its size, whereas an expression $\mathsf{old}(e)$ only denotes one particular state location. The absence of powerful specification constructs like the finite sequences in our specification language also seems to be the cause of the incompleteness of Abadi and Leino's logic for reasoning about object-oriented programs [AL03].

We have also investigated whether our techniques can be used to reason about open programs. We have made use of behavioral subtyping in Chapter 8 to turn our proof outline logic into a modular logic that is suitable for open programs. But we have also argued that behavioral subtyping is *not* the answer to all modularity issues. It is only a means to obtain a *sound* modular logic for open programs. It does not tell us in which sense a program logic for open programs can still be complete. We have given a possible answer to that question in Section 8.3. The completeness notion that we have proposed there is tailored to the usual method specifications that describe initial and final states of method executions.

However, stronger specification techniques must be developed in order to fully master the extensible nature of object-oriented programs. Methods should additionally be able to specify how their behavior can be influenced by future program extensions. To this end, methods have to reveal more details of their internal behavior. We have illustrated this necessity in Section 8.4. Integrating trace specifications (cf. [SF04]) into our logic appears to be a valuable step towards a definite solution for reasoning about extensible object-oriented programs.

Bibliography

[Á05] Erika Ábrahám. *An Assertional Proof System for Multithreaded Java: Theory and Tool Support*. PhD thesis, Leiden University, 2005.

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[AdB90] Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–162, 1990.

[AdB94] Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.

[AdBdRS03] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof outlines for monitors in Java. In *Formal Methods for Open Object-Based Distributed Systems (Proc. of FMOODS 2003)*, volume 2884 of *LNCS*, pages 155–169. Springer, 2003.

[AL03] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 11–41. Springer, 2003.

[AMdBdRS02] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In *Foundations of Software Science and Computation Structures (Proc. of FOSSACS 2002)*, volume 2303 of *LNCS*, pages 5–20, 2002.

[Ame87] Piere America. Inheritance and subtyping in a parallel object-oriented language. In *Proc. of the European Conference on*

Object-Oriented Programming, volume 276 of *LNCS*, pages 234–242. Springer, 1987.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 60–90. Springer, 1991.

[Apt81] Krzysztof R. Apt. Ten Years of Hoare's Logic: A Survey - Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[BMR95] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[BMW89] A. Bijlsma, P.A. Matthews, and J.G. Wiltink. A sharp proof rule for procedures in wp semantics. *Acta Informatica*, 26:409–419, 1989.

[BN04] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction (MPC 2004)*, volume 3125 of *LNCS*, pages 54–84, 2004.

[BNSS04] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *Formal techniques for Java-like Programs (Proceedings of the ECOOP Workshop FTfJP '2004)*, 2004. The proceedings appeared as technical report nr. NIII-R0426, University of Nijmegen, 2004.

[BP03] Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs. Proceedings of the ECOOP 2003 Workshop*, pages 51–60, 2003. Technical Report 408, ETH Zurich.

[BRL03] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *Proc. of Formal Methods (FME 2003)*, volume 2805 of *LNCS*, pages 422–439, 2003.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages*. The MIT Press, 2002.

[CC00] Yonghao Chen and Betty H. C. Cheng. A semantic foundation for specification matching. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 5, pages 91–109. Cambridge University Press, 2000.

[CK05] David R. Cok and Joseph R. Kiniry. ESC/Java2: Unitying ESC/Java and JML. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.

[CL02] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, 2002.

[CO81] Robert Cartwright and Derek Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.

[COB03] Cristiano Calcagno, Peter O'Hearn, and Richard Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(2):557–581, 2003.

[Cok04] David R. Cok. Reasoning with specifications containing method calls in jml and first-order provers. In *Formal techniques for Java-like Programs (Proceedings of the ECOOP Workshop FTfJP '2004)*, 2004. The proceedings appeared as technical report nr. NIII-R0426, University of Nijmegen.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *Siam Journal of Computing*, 7(1):70–90, February 1978.

[CPN98] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, 1998.

[dB80] J.W. de Bakker. *Mathematical theory of program correctness*. Prentice-Hall, 1980.

[dB91] F.S. de Boer. *Reasoning about dynamically evolving process structures*. PhD thesis, Vrije Universiteit, 1991.

[dB99] Frank S. de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures (FOSSACS '99)*, volume 1578 of *LNCS*, pages 135–149. Springer, 1999.

[dB02] F. S. de Boer. A hoare logic for dynamic networks of asychronously communicating deterministic processes. *Theoretical Computer Science*, 274(1–2):3–41, 2002.

[dBP04] Frank S. de Boer and Cees Pierik. How to Cook a complete Hoare logic for your pet OO language. In *Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *LNCS*, pages 111–133. Springer, 2004.

[DEK99] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE '96)*, pages 258–267. IEEE Computer Society Press, 1996.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[DVE00] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited. Available from http://slurp.doc.ic.ac.uk/pubs.html, September 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of Programming Language Design and Implementation (PLDI 2002)*, pages 234–245. ACM Press, 2002.

[Flo67] Robert W. Floyd. Assigning meaning to programs. In *Proc. Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[Gor75] G.A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dep. Computer Science, Univ. Toronto, 1975.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. S. E. Maibaum, editor, *FASE 2000*, volume 1783 of *LNCS*, pages 284–303, 2000.

[HK00] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 208–221, 2000.

[HM03] Peter V. Homeier and David F. Martin. Secure mechanical verification of mutually recursive procedures. *Information and Computation*, 187:1–19, 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Proc of the Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116, 1971.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hui01] Marieke Huisman. *Reasoning about Java programs in higher order logic with PVS and Isabelle*. PhD thesis, Katholieke Universiteit Nijmegen, 2001.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 132–146. ACM Press, 1999.

[Jac03] Bart Jacobs. Java's integral types in PVS. In *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *LNCS*, pages 1–15. Springer, 2003.

[Jac04] Bart Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *The Journal of Logic and Algebraic Programming*, 58:61–88, 2004.

[JKW03] Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In *Formal Methods for Components and Objects (Proc. of FMCO 2002)*, volume 2852 of *LNCS*, pages 202–219. Springer, 2003.

[JMR04] Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology and Software Technology (Proc. of AMAST 2004)*, volume 3116 of *LNCS*, pages 241–257. Springer, 2004.

[Kle99] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11:541–566, 1999.

[Kow77] Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.

[LBR04] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Department of Computer Science, Iowa State University, June 2004.

[LCC+02] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *Formal Methods for Components and Objects (Proc. of FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284. Springer, 2002.

[LD00] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proc.of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1998)*, volume 33 of *ACM SIGPLAN Notices*, pages 144–153, 1998.

[LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.

[LM05] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Formal Methods (FM 2005)*, LNCS. Springer, 2005. In this volume.

[LN02] K. Rustan. M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

[LPHZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI 2002)*, pages 246–257. ACM Press, 2002.

[LS79] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, 1979.

[LS04] K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 218–227. IEEE, 2004.

[LW90] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes. In *OOPSLA/ECOOP '90 Proceedings*, ACM SIGPLAN Notices, pages 212–223. ACM Press, 1990.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995.

[Mey85] J.J.Ch. Meyer. *Programming Calculi Based on Fixed Point Transformations: Semantics and Applications*. PhD thesis, Vrije Universiteit, 1985.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mok03] Heng Ngee Mok. *From Java to C#: A Developer's Guide*. Addison-Wesley, 2003.

[MPH00] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 36–77. Springer, 2000.

[MPMU04] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[Mül02] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[Nau00] David A. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 7:201–208, 2000.

[Nau05] David A. Naumann. Observational purity and encapsulation. In *Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 190–204. Springer, 2005.

[NB04] David A. Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *Proc. of Logic in Computer Science (LICS 2004)*, pages 313–323. IEEE, 2004.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Principles of Programming Languages (Proc. of POPL 1998)*, pages 161–170. ACM Press, 1998.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Old83] Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.

[PCdB04] Cees Pierik, Dave Clarke, and Frank S. de Boer. Creational invariants. In *Formal techniques for Java-like Programs (Proceedings of the ECOOP Workshop FTfJP 2004)*, 2004. The proceedings appeared as technical report nr. NIII-R0426, University of Nijmegen, 2004.

[PCdB05] Cees Pierik, Dave Clarke, and Frank S. de Boer. Controlling object allocation using creation guards. In *FM 2005: Formal Methods*, volume 3582 of *LNCS*, pages 59–74. Springer, 2005.

[PdB03a] Cees Pierik and Frank S. de Boer. A rule of adaptation for OO. Technical Report UU-CS-2003-032, Institute of Information and Computing Sciences, Utrecht University, The Netherlands, October 2003.

[PdB03b] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pages 64–78, 2003.

[PdB03c] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute of Information and Computing Sciences, Utrecht University, The Netherlands, March 2003.

[PdB04] Cees Pierik and Frank S. de Boer. Modularity and the rule of adaptation. In *Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *LNCS*, pages 394–408. Springer, 2004.

[PdB05a] Cees Pierik and Frank S. de Boer. On behavioral subtyping and completeness. In *7th Workshop on Formal techniques for Java-like Programs (FTfJP 2005)*, 2005.

[PdB05b] Cees Pierik and Frank S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[PHM98] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods*, pages 404–423. Springer, 1998.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Proc. of the European Symposium on Programming (ESOP 1999)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Rot05] Andreas Roth. Specification and verification of encapsulation in Java programs. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005)*, volume 3535 of *LNCS*, pages 195–210. Springer, 2005.

[RWH01] Bernhard Reus, Martin Wirsing, and Rolf Hennicker. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Heinrich Hussmann, editor, *FASE 2001*, volume 2029 of *LNCS*, pages 300–317, 2001.

[SF99] Neelam Soundarajan and Stephen Fridella. Enriching behavioral subtyping. Unpublished, June 1999.

[SF04] Neelam Soundarajan and Stephen Fridella. Incremental reasoning for object-oriented systems. In *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 302–333. Springer, 2004.

[Smi92] G. Smith. *An Object-Oriented Approach to Formal Specifications*. PhD thesis, University of Queensland, October 1992.

[Smi00] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 38–45, 1986.

[Spi92] J.M. Spivey. *The Z notation: A reference manual*. Prentice-Hall, second edition, 1992.

[Ste04] Kurt Stenzel. A formally verified calculus for full Java Card. In *Algebraic Methodology and Software Technology*, volume 3116 of *LNCS*, pages 491–505. Springer, 2004.

[TZ88] J.V. Tucker and J.I Zucker. *Program correctness over abstract data types with error-state semantics*, volume 6 of *CWI Monographs*. North-Holland, 1988.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001.

[vO01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[ZHL+96] Job Zwiers, Ulrich Hannemann, Yassine Lakhnech, Willem-Paul de Roever, and Frank Stomp. Modular completeness: Integrating the reuse of specified software in top-down program development. In *Proc. of Formal Methods Europe (FME 1996)*, volume 1051 of *LNCS*, pages 595–608. Springer, 1996.

[Zwi87] J. Zwiers. *Compositionality, Concurrency, and Partial Correctness*, volume 321 of *LNCS*. Springer-Verlag, 1987.

Samenvatting

Steeds meer taken worden tegenwoordig uitgevoerd door computerprogramma's. Niet zelden betreft het taken waarvan grote economische of maatschappelijke belangen afhangen. Deze belangen rechtvaardigen een grote inspanning om de correcte werking van dergelijke programma's aan te tonen.

De correctheid van een programma kan aangetoond worden door middel van bewijsschetsen voor de procedures waaruit het programma bestaat. Een bewijsschets van een procedure beschrijft met formules uit een formele taal de gewenste begin- en eindtoestanden van de procedure. Verder beschrijft het de mogelijke toestanden op belangrijke punten in de implementatie van de procedure. Zo wordt op een beknopte manier getoond waarom een procedure aan zijn specificatie voldoet. Een geldige bewijsschets beschrijft de wezenlijke elementen van een sluitend wiskundig bewijs voor de correctheid van een procedure.

Een bewijsschets is echter niet zondermeer geldig. De juistheid van een bewijsschets kan aangetoond worden door de geldigheid van een verzameling additionele bewijsverplichtingen te bewijzen. Deze bewijsverplichtingen kunnen berekend worden op basis van de implementatie van de procedure en de formules in de bewijsschets. Doorgaans wordt er vervolgens een stellingbewijzer gebruikt om na te gaan of de bewijsverplichtingen geldig zijn.

In dit proefschrift wordt beschreven hoe de additionele bewijsverplichtingen van bewijsschetsen van objectgeoriënteerde programma's kunnen worden berekend. Het proefschrift definieert voor dit doel een formele calculus. Deze calculus houdt rekening met belangrijke objectgeoriënteerde kenmerken als overerving, subtype polymorfisme en dynamische binding. Er is gestreefd de calculus geschikt te maken voor de objectgeoriënteerde eigenschappen van de populaire programmeertalen Java en C#.

De ontwikkelde calculus heeft een aantal belangrijke kenmerken. Allereerst is de calculus geschikt voor bewijsschetsen waarvan de specificatietaal is gebaseerd op de expressies uit de programmeertaal. Dergelijke bewijsschetsen zijn eenvoudiger te begrijpen en op te stellen door programmeurs. Bovendien zijn de berekende bewijsverplichtingen zelf ook formules uit dezelfde specificatietaal. Verder kan de calculus volledig mechanisch toegepast worden. Tenslotte wordt in dit proefschrift aangetoond dat de berekende bewijsverplichtingen een sluitende bewijs opleveren voor de corresponderende bewijsschets, en dat de correctheid van iedere geldige bewijsschets met behulp van deze calculus aange-

toond kan worden (voor zover dit van de calculus afhangt).

De genoemde calculus wordt beschreven in de eerste hoofdstukken van dit proefschrift. De laatste drie hoofdstukken beschrijven uitbreidingen van de calculus en een applicatie waarmee de ontwikkelde technieken toegepast kunnen worden op Java programma's.

Hoofdstuk 2 geeft een overzicht van de belangrijkste kenmerken van objectgeoriënteerde programmeertalen en definieert een eenvoudige taal waarin al deze eigenschappen samengebracht zijn. Voor deze taal wordt een formele semantiek gegeven, zodat de betekenis van alle taalconstructen eenduidig vastgelegd is.

Het volgende hoofdstuk bevat een formele definitie van bewijsschetsen. Maar eerst wordt de specificatietaal beschreven waarin de onderdelen van bewijsschetsen uitgedrukt worden. Zowel de taal als de bewijsschetsen krijgen een formele semantiek.

In het vierde tot en met het zesde hoofdstuk wordt de calculus opgebouwd aan de hand van de verschillende taalconstructen in objectgeoriënteerde programmeertalen. Allereerst worden in Hoofdstuk 4 toekenningen behandeld. Voor toekenningen aan lokale variabelen en voor toekenningen aan velden van objecten wordt een zwakste preconditie calculus gedefinieerd. Uitgelegd wordt hoe omgegaan kan worden met het verschijnsel dat objecten meerdere gelijknamige velden kunnen hebben door overerving. Ook het beruchte alias probleem wordt behandeld.

Hoofdstuk 5 is een centraal hoofdstuk van dit proefschrift. Het introduceert een nieuwe manier om bewijsverplichtingen te genereren voor aanroepen van procedures in objectgeoriënteerde programma's. Voor dit doel wordt een zogenaamde aanpassingsregel (*adaptation rule*) ontwikkeld. Ook wordt uitgebreid beschreven waarom een dergelijke regel noodzakelijk is voor de mechanische generatie van bewijsverplichtingen in de context van bewijsschetsen.

Hoofdstuk 6 beschrijft het laatste onderdeel van de basiscalculus: een zwakste preconditie calculus voor object allocatie. Ook wordt ingegaan op de generatie van bewijsverplichtingen voor de daarop volgende initialisatiefase van objecten.

Hoofdstuk 7 rondt het eerste deel van het proefschrift af. Het begint met een samenvatting van de ontwikkelde calculus. Daarna volgen er bewijzen van twee belangrijke eigenschappen van de calculus. Eerst wordt aangetoond dat de berekende bewijsverplichtingen een sluitende bewijs opleveren voor de corresponderende bewijsschets. Vervolgens wordt bewezen dat de correctheid van iedere geldige bewijsschets met behulp van de calculus aangetoond kan worden.

De hierboven genoemde technieken zijn geschikt voor bewijsschetsen van gesloten programma's; dat zijn programma's waarvan alle code beschikbaar is. In Hoofdstuk 8 worden de problemen rond programma-uitbreidingen behandeld. Het hoofdstuk begint met een formele definitie van *behavioral subtyping* in bewijsschetsen. Vervolgens wordt een nieuwe manier beschreven waarmee nagegaan kan worden of een bepaalde klasse gebruikt kan worden als *behavioral* subtype van een andere klasse. Daarna worden nieuwe bewijsverplichtingen gegeven voor aanroepen van procedures. Met deze uitbreidingen wordt de calculus

geschikt voor het modulair controleren van bewijsschetsen van open, uitbreidbare programma's.

Tot op heden was niet duidelijk aan welke volledigheidseisen een calculus voor open programma's moet voldoen. Hoofdstuk 8 eindigt om die reden met een nieuwe volledigheidsdefinitie. Bovendien wordt aan de hand van een voorbeeldprogramma een probleem geschetst dat met de huidige technieken voor bewijsschetsen van open programma's nog niet opgelost kan worden.

Hoofdstuk 9 introduceert een nieuwe techniek die ondersteuning biedt voor invarianten die gevoelig zijn de allocatie van nieuwe objecten. Uitgelegd wordt hoe additionele formules die moeten gelden bij de allocatie van een nieuw object kunnen garanderen dat een invariant niet ongemerkt geschonden wordt.

Tenslotte wordt in het tiende hoofdstuk een applicatie beschreven waarmee de technieken die in dit proefschrift beschreven staan, toegepast kunnen worden op bewijsschetsen van Java programma's. De applicatie ondersteunt de gebruiker bij het opstellen van de bewijsschetsen en berekent de corresponderende bewijsverplichtingen. Vervolgens wordt er een stellingbewijzer gebruikt om de geldigheid van de bewijsverplichtingen te onderzoeken. De resultaten van de stellingbewijzer worden uiteindelijk getoond aan de gebruiker.

Dankwoord

Het schrijven van een proefschrift lijkt wel wat op het maken van een trektocht door de Andes: je wordt beloond met prachtige vergezichten die je de inspanningen van de tocht steeds weer doen vergeten. En als je met je reisgenoten de eindbestemming haalt, bedank je elkaar hartelijk voor de goede tijd die je met elkaar gehad hebt. Je vertelt elkaar dan wat jij het mooiste gedeelte van de tocht vond. En als je eenmaal weer thuis bent, worden de herinneringen steeds mooier en de verhalen steeds spannender

Mijn belangrijkste reisgenoot tijdens het hele promotietraject was Frank de Boer. Als begeleider heeft hij de hele tocht van dichtbij meegemaakt. Zijn positieve instelling en enthousiasme hebben mij vaak op cruciale momenten het juist duwtje in de rug gegeven. Daarvoor ben ik hem veel dank verschuldigd.

Een even plezierige reisgenoot was mijn promotor John-Jules Meyer. De bijeenkomsten die ik met hem en Frank had, liepen altijd ongemerkt uit. Nooit waren we op tijd klaar voor de lunch. De leesbaarheid van mijn proefschrift is sterk verbeterd door zijn adviezen. Bovendien wist hij vaak interessante verbanden aan te wijzen tussen mijn onderzoek en dat van veel andere informatici.

De leden van de leescommissie van mijn proefschrift wil ik bedanken voor hun medewerking. Zij mogen hier niet ongenoemd blijven: Farhad Arbab, Ernst-Rüdiger Olderog, Wishnu Prasetya, Willem-Paul de Roever en Job Zwiers.

Met veel genoegen denk ik terug aan de bijeenkomsten in het kader van het MobiJ-project. Ik kreeg er een kijkje in de keuken van het internationale onderzoek. En 's avonds werden er mooie verhalen verteld in allerlei Duitse eetgelegenheden.

Ik wil ook al mijn collega's van de *Intelligent Systems* groep bedanken voor de goede contacten. Tijdens onze gezamelijke lunches was er altijd iets om over te praten. Ook van de intensieve discussies heb ik veel geleerd. In het bijzonder wil ik mijn kamergenoten Paul Harrenstein en Birna van Riemsdijk bedanken voor de vele goede gesprekken. Paul stond ook altijd klaar om mij over alle LaTeX-hobbels op mijn pad te helpen.

Aalt-Jan van Dijk en Jan Reitsma waren meteen bereid mij als paranimf terzijde te staan. Dat wil ik nu alvast met dank vermelden. Ik wens ze ook allebei succes toe bij het voltooien van hun eigen proefschriften.

Aan mijn beide ouders heb ik veel te danken. Zij hebben mij tijdens mijn studiejaren (en lang daarvoor) altijd gesteund en leven ook nu nog op allerlei

manieren mee. Zulke ouders durf ik iedereen toe te wensen.

En dan is er nog één reisgenoot over. Ook zij heeft van heel dichtbij mijn tocht meegemaakt. Ze was op veel momenten onmisbaar. Met haar hoop ik nog lang mijn weg te vervolgen. Natuurlijk heb ik het dan over mijn lieve echtgenote Anne Marie.

Curriculum Vitae

Naam:	Cees Pierik
Geboren:	20 november 1978 te Hasselt

Opleidingen

1991-1997	VWO, scholengemeenschap Pieter Zandt, Kampen
1997-2001	Informatica, Universiteit Utrecht (cum laude)

Functies

2002 - 2005	assistent in opleiding, Instituut voor Informatica en Informatiekunde (Universiteit Utrecht)
2006 - heden	consultant, LogicaCMG

SIKS Dissertation Series

1998

Johan van den Akker, *DEGAS - An Active, Temporal Database of Autonomous Objects*, CWI, 1998-1

Floris Wiesman, *Information Retrieval by Graphically Browsing Meta-Information*, UM, 1998-2

Ans Steuten, *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*, TUD, 1998-3

Dennis Breuker, *Memory versus Search in Games*, UM, 1998-4

E.W. Oskamp, *Computerondersteuning bij Straftoemeting*, RUL, 1998-5

1999

Mark Sloof, *Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products*, VU, 1999-1

Rob Potharst, *Classification using decision trees and neural nets*, EUR, 1999-2

Don Beal, *The Nature of Minimax Search*, UM, 1999-3

Jacques Penders, *The practical Art of Moving Physical Objects*, UM, 1999-4

Aldo de Moor, *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*, KUB, 1999-5

Niek J.E. Wijngaards, *Re-design of compositional systems*, VU, 1999-6

David Spelt, *Verification support for object database design*, UT, 1999-7

Jacques H.J. Lenting, *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.*, UM, 1999-8

2000

Frank Niessink, *Perspectives on Improving Software Maintenance*, VU, 2000-1

Koen Holtman, *Prototyping of CMS Storage Management*, TUE, 2000-2

Carolien M.T. Metselaar, *Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief*, UVA, 2000-3

Geert de Haan, *ETAG, A Formal Model of Competence Knowledge for User Interface Design*, VU, 2000-4

Ruud van der Pol, *Knowledge-based Query Formulation in Information Retrieval*, UM, 2000-5

Rogier van Eijk, *Programming Languages for Agent Communication*, UU, 2000-6

Niels Peek, *Decision-theoretic Planning of Clinical Patient Management*, UU, 2000-7

Veerle Coup, *Sensitivity Analyis of Decision-Theoretic Networks*, EUR, 2000-8

Florian Waas, *Principles of Probabilistic Query Optimization*, CWI, 2000-9

Niels Nes, *Image Database Management System Design Considerations, Algorithms and Architecture*, CWI, 2000-10

Jonas Karlsson, *Scalable Distributed Data Structures for Database Management*, CWI, 2000-11

2001

Silja Renooij, *Qualitative Approaches to Quantifying Probabilistic Networks*, UU, 2001-1

Koen Hindriks, *Agent Programming Languages: Programming with Mental Models*, UU, 2001-2

Maarten van Someren, *Learning as problem solving*, UvA, 2001-3

Evgueni Smirnov, *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*, UM, 2001-4

Jacco van Ossenbruggen, *Processing Structured Hypermedia: A Matter of Style*, VU, 2001-5

Martijn van Welie, *Task-based User Interface Design*, VU, 2001-6

Bastiaan Schonhage, *Diva: Architectural Perspectives on Information Visualization*, VU, 2001-7

Pascal van Eck, *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*, VU, 2001-8

Pieter Jan 't Hoen, *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*, RUL, 2001-9

Maarten Sierhuis, *Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design*, UvA, 2001-10

Tom M. van Engers, *Knowledge Management: The Role of Mental Models in Business Systems Design*, VUA, 2001-11

2002

Nico Lassing, *Architecture-Level Modifiability Analysis*, VU, 2002-01

Roelof van Zwol, *Modelling and searching web-based document collections*, UT, 2002-02

Henk Ernst Blok, *Database Optimization Aspects for Information Retrieval*, UT, 2002-03

Juan Roberto Castelo Valdueza, *The Discrete Acyclic Digraph Markov Model in Data Mining*, UU, 2002-04

Radu Serban, *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents*, VU, 2002-05

Laurens Mommers, *Applied legal epistemology; Building a knowledge-based ontology of the legal domain*, UL, 2002-06

Peter Boncz, *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*, CWI, 2002-07

Jaap Gordijn, *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*, VU, 2002-08

Willem-Jan van den Heuvel, *Integrating Modern Business Applications with Objectified Legacy Systems*, KUB, 2002-09

Brian Sheppard, *Towards Perfect Play of Scrabble*, UM, 2002-10

Wouter C.A. Wijngaards, *Agent Based Modelling of Dynamics: Biological and Organisational Applications*, VU, 2002-11

Albrecht Schmidt, *Processing XML in Database Systems*, UVA, 2002-12

Hongjing Wu, *A Reference Architecture for Adaptive Hypermedia Applications*, TUE, 2002-13

Wieke de Vries, *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*, UU, 2002-14

Rik Eshuis, *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*, UT, 2002-15

Pieter van Langen, *The Anatomy of Design: Foundations, Models and Applications*, VU, 2002-16

Stefan Manegold, *Understanding, Modeling, and Improving Main-Memory Database Performance*, UVA, 2002-17

2003

Heiner Stuckenschmidt, *Onotology-Based Information Sharing In Weakly Structured Environments*, VU, 2003-1

Jan Broersen, *Modal Action Logics for Reasoning About Reactive Systems*, VU, 2003-02

Martijn Schuemie, *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*, TUD, 2003-03

Milan Petkovic, *Content-Based Video Retrieval Supported by Database Technology*, UT, 2003-04

Jos Lehmann, *Causation in Artificial Intelligence and Law - A modelling approach*, UVA, 2003-05

Boris van Schooten, *Development and specification of virtual environments*, UT, 2003-06

Machiel Jansen, *Formal Explorations of Knowledge Intensive Tasks*, UvA, 2003-07

Yongping Ran, *Repair Based Scheduling*, UM, 2003-08

Rens Kortmann, *The resolution of visually guided behaviour*, UM, 2003-09

Andreas Lincke, *Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture*, UvT, 2003-10

Simon Keizer, *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*, UT, 2003-11

Roeland Ordelman, *Dutch speech recognition in multimedia information retrieval*, UT, 2003-12

Jeroen Donkers, *Nosce Hostem - Searching with Opponent Models*, UM, 2003-13

Stijn Hoppenbrouwers, *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*, KUN, 2003-14

Mathijs de Weerdt, *Plan Merging in Multi-Agent Systems*, TUD, 2003-15

Menzo Windhouwer, *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses*, CWI, 2003-16

David Jansen, *Extensions of Statecharts with Probability, Time, and Stochastic Timing*, UT, 2003-17

Levente Kocsis, *Learning Search Decisions*, UM, 2003-18

2004

Virginia Dignum, *A Model for Organizational Interaction: Based on Agents, Founded in Logic*, UU, 2004-01

Lai Xu, *Monitoring Multi-party Contracts for E-business*, UvT, 2004-02

Perry Groot, *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*, VU, 2004-03

Chris van Aart, *Organizational Principles for Multi-Agent Architectures*, UVA, 2004-04

Viara Popova, *Knowledge discovery and monotonicity*, EUR, 2004-05

Bart-Jan Hommes, *The Evaluation of Business Process Modeling Techniques*, TUD, 2004-06

Elise Boltjes, *Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes*, UM, 2004-07

Joop Verbeek, *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise*, UM, 2004-08

Martin Caminada, *For the Sake of the Argument; explorations into argument-based reasoning*, VU, 2004-09

Suzanne Kabel, *Knowledge-rich indexing of learning-objects*, UVA, 2004-10

Michel Klein, *Change Management for Distributed Ontologies*, VU, 2004-11

The Duy Bui, *Creating emotions and facial expressions for embodied agents*, UT, 2004-12

Wojciech Jamroga, *Using Multiple Models of Reality: On Agents who Know how to Play*, UT, 2004-13

Paul Harrenstein, *Logic in Conflict. Logical Explorations in Strategic Equilibrium*, UU, 2004-14

2004-15, *UU*, Arno Knobbe, 2004-14 Multi-Relational Data Mining

Federico Divina, *Hybrid Genetic Relational Search for Inductive Learning*, VU, 2004-16

Mark Winands, *Informed Search in Complex Games*, UM, 2004-17

Vania Bessa Machado, *Supporting the Construction of Qualitative Knowledge Models*, UvA, 2004-18

Thijs Westerveld, *Using generative probabilistic models for multimedia retrieval*, UT, 2004-19

Madelon Evers, *Learning from Design: facilitating multidisciplinary design teams*, Nyenrode, 2004-20

2005

Floor Verdenius, *Methodological Aspects of Designing Induction-Based Applications*, UVA, 2005-01

Erik van der Werf, *AI techniques for the game of Go*, UM, 2005-02

Franc Grootjen, *A Pragmatic Approach to the Conceptualisation of Language*, RUN, 2005-03

Nirvana Meratnia, *Towards Database Support for Moving Object data*, UT, 2005-04

Gabriel Infante-Lopez, *Two-Level Probabilistic Grammars for Natural Language Parsing*, UVA, 2005-05

Pieter Spronck, *Adaptive Game AI*, UM, 2005-06

Flavius Frasincar, *Hypermedia Presentation Generation for Semantic Web Information Systems*, TUE, 2005-07

Richard Vdovjak, *A Model-driven Approach for Building Distributed Ontology-based Web Applications*, TUE, 2005-08

Jeen Broekstra, *Storage, Querying and Inferencing for Semantic Web Languages*, VU, 2005-09

Anders Bouwer, *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*, UVA, 2005-10

Elth Ogston, *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*, VU, 2005-11

Csaba Boer, *Distributed Simulation in Industry*, EUR, 2005-12

Fred Hamburg, *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*, UL, 2005-13

Borys Omelayenko, *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*, VU, 2005-14

Tibor Bosse, *Analysis of the Dynamics of Cognitive Processes*, VU, 2005-15

Joris Graaumans, *Usability of XML Query Languages*, UU, 2005-16

Boris Shishkov, *Software Specification Based on Re-usable Business Components*, TUD, 2005-17

Danielle Sent, *Test-selection strategies for probabilistic networks*, UU, 2005-18

Michel van Dartel, *Situated Representation*, UM, 2005-19

Cristina Coteanu, *Cyber Consumer Law, State of the Art and Perspectives*, UL, 2005-20

Wijnand Derks, *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*, UT, 2005-21

2006

Samuil Angelov, *Foundations of B2B Electronic Contracting*, TUE, 2006-01

Cristina Chisalita, *Contextual issues in the design and use of information technology in organizations*, VU, 2006-02

Noor Christoph, *The role of metacogniti-*

ve skills in learning to solve problems, UVA, 2006-03

Marta Sabou, *Building Web Service Ontologies*, VU, 2006-04

Cees Pierik, *Validation Techniques for Object-Oriented Proof Outlines*, UU, 2006-05