Formal Semantic Specification of a Core Object-Oriented Language

Diploma Thesis

Thomas Bietenhader

Supervised by
Prof. Dr. Bertrand Meyer
Bernd Schoeller

Chair of Software Engineering
Department of Computer Science
ETH Zürich

September 2004

Abstract

In this project a formal semantic definition for a small object-oriented language is developed, in the context of formal verification of program correctness. From its inherent modular nature, its ability to support abstraction via information hiding and subtyping, the object-oriented paradigm seems predestined for efficient reuse of software components. This raises hopes to apply these advantages also on the topic of verification. Problems arising from this enterprise are discussed and an overview on the actual situation of research is given. A conclusion of this discussion is that specifications become the most critical part in modular verification.

Contents

Chapter 1

Introduction

The aim of this project is to develop a formal semantic definition of a small, object-oriented programming language. The need for a formal semantics arises e.g. in the context of automated program verification. The interest in program correctness is also the actual motive for this project. In order to be automatically checked, proofs also have to be stated formally. Thus besides a formal definition of the semantics also an adequate calculus to express proofs has to be available. However such a calculus is not provided here. The guiding principle in program correctness can be stated as follows:

> "Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs."
>
> *John McCarthy* [4]

In this chapter first programming is introduced as a method for problem solving. Requirements to a language that supports program verification is discussed briefly. An overview on object-oriented concepts follows. The modular structure and the degree of abstraction it permits make the object-oriented paradigm a promising candidate to achieve the goal of efficient program verification. Modular verification raises theoretical problems which are primarily related to the issue of specification. Approaches suggested by various researches to overcome these problems are sketched. The chapter concludes with an outlook on the rest of the work.

1.1 Motivation

1.1.1 Programming Languages

In the software construction process, the task of formulating the solution in a (high-level) programming language is the last one, except validation and documentation. The preceding tasks have to transform the original specification into a form so that the problem can be stated and solved in terms of a programming language. In reality these stages cannot be strictly separated. Moreover the process of implementation itself may consist of a sequence of refinement steps, according to Wirth's [15] approach of *stepwise refinement*. The key issue is that prior writing code one has to have a problem specification adequate to the chosen programming language.

Ideally the specification on the programming language level can be expressed in the language itself. The *Design by Contract* [7] method is an important step toward programs which not only contain the implementation but also (parts) of the specification. On one hand this eases reuse, as a client of a foreign program module can obtain information about the module's functionality. On the other hand the built-in specification can be used for automated program verification.

In the end a programming language is nothing else than a formal system, designed to make automated generation of computer executable code possible. It is in the nature of a formal system to impose constraints on how to formulate things. These constraints can roughly be divided into two kinds:

1. *Syntactical* constraints are related to the form a program has to be written in. The syntax usually is defined through a *grammar*, defining the rules according to which programs have to be composed.

2. *Semantical* constraints are related to the contents or meaning of a program. Syntactical correctness does not imply that a program has a meaning. A typical semantical constraint in strongly typed languages is well-typedness of programs.

The topic of syntactical correctness will not be pursued further in this project. For simplicity the term 'program' always refers to a syntactically correct program.

The art of programming lies in using the formalism provided by a language so that the resulting program corresponds to its abstract specification – stated simply this means to get the program correct. And correctness is the most important property of a program. So programming should be concerned with:

> "How to write correct programs and know it."
>
> *Harlan D. Mills* [10]

Besides correctness the issues reusability and extendability of software are important. These properties allow to develop software efficiently, since one can concentrate on new aspects of a problem and does not have to solve the same problems again and again. Relying on a correct standard solution has an invaluable impact on reliability. A component which is known to be correct will stay correct when reused.

Extendability is the property that to an existing system or set of modules new modules can be added without changing the already existing parts. Hence in an extendable system a module does not have to know its environment completely to work correctly. This property is crucial for efficient reuse also. If correctness were guaranteed only for fixed environments, reuse of a standard solution would not imply correctness. Rather one had to verify correctness anew in every new context. How to state correct solutions with a limited knowledge of the context so that it stays correct in every possible context turns out to be the major challenge of programming, or better specification in general. A final answer to this problem is not yet found.

As requirements to a programming language we may state so far that it has to support a programmer in writing correct programs. Ideally some of the checking is done automatically, as is done for types in typed languages. Furthermore it should have facilities which allow efficient reuse and extension. For a detailed discussion of software quality properties see [6].

1.1.2 Semantics

The main purpose of a programming language is to describe systems or problem solutions in an abstract way. As this description is stated in a formal way, its meaning is not self-explanatory. Hence beside the syntactic rules how to form programs also rules which map programs to a meaning are needed. Stated differently this means to translate programs into a language which already has a (natural) meaning.

The rules how the meaning of a programs is determined are stated in the semantic definition of the language. The semantics of a programming language is typically defined by mathematical models.

Such a model could e.g. describe an abstract computer where the store is represented by a set of memory locations. The state then is described by a function on the store which for each location yields the current value stored. A program finally could be interpreted as a function which transforms state.

Thus a formal semantic definition of a programming language will first define a mathematical model. Then for each language construct an interpretation in terms of the model has to be given. In the end a given program is described completely by the mathematical model.

Theoretical Limits on Proof Automation

From a theoretical point of view a programming language can be considered a computational model, as e.g. *Turing Machines* or *Markov Algorithms* are. A language which provides a computational power equivalent to that of a Turing Machine underlies also its computational limitations. One of the most famous limitations is the undecidability of the *halting problem.* According to this the problem whether a program will ever terminate on a given input cannot be decided algorithmically. Since the problem of program correctness can be reduced to the halting problem there does not exist a program which can prove or disprove correctness of programs. The best one can do is to verify existing proofs automatically. Finding proofs is an act of creativity in principle. However proof assistants can be used to support a human proofer. So at best one can hope for semi-automated proof strategies.

1.2 A Glance at Object-Orientation

An exact definition of object-orientation does not seem to exist. We will concentrate on concepts which are widely accepted to be indispensable for the object-oriented paradigm. These are the *modular structure*, *information hiding* and *inheritance*, although these concepts are not exclusive to the object-oriented paradigm, neither can it claim to have introduced it. Also the role of *objects* has to be clarified, since the name of the paradigm suggests that they are central to object-orientation. For a more detailed discussion we refer to Meyer's treatment in [6], in particular chapter 2.

1.2.1 Classes

The *class* is the main concept for structuring programs. A class is an autonomous unit or module which defines an *abstract data type.* An abstract data type consists of a domain of values together with a set of operations defined on these values. In connection with object-orientation these operations are often called the *methods* of a class. However, following Meyer's terminology we will refer to them as *features.*

In most object-oriented languages a class may provide a full implementation of the data type, or only a partial implementation or none at all. In the first case the class is called *concrete* or *effective.* In the latter cases it is called *abstract* or *deferred.* Deferred classes are useful to describe a behavior that may be shared by different types, but which is too abstract as that a reasonable implementation could be given. An example is the concept of comparability on ordered domains, e.g. natural numbers. There are many abstractions whose values can be ordered. But an abstraction whose only property is that the values can be ordered is hard to find.

To every effective class there exists a set of objects, called *instances* of the class. An object is a runtime entity which represents a value of the abstract data type during program execution. Its behavior is determined by the features of the corresponding class it belongs to.

An object which is an instance of some class C can represent any value of the abstract data type defined by the class C. So the state of an instance is the value it represents. Besides this an object has an identity of its own, independent of what its current state is. Thus given two objects one has to distinguish whether they are the same object or just represent the same abstract value.

Features play a similar role as procedures in non-object-oriented imperative languages. The difference to procedures is that features are always executed relative to a designated object, called the *target* of an invocation. In object-oriented programming computation is based on feature invocations *on* objects, rather than procedure calls taking a list of arguments. In the former case one can consider the target object to be responsible for the effect of the invocation. While in the latter case the called procedure is responsible to attain the intended effect. So there is a shift from procedures to objects. In order to accomplish a task one first has to determine the responsible object and then the appropriate feature to be invoked. The term object-oriented denotes this shift which makes objects the main actors of computation.

1.2.2 Information Hiding

An abstract specification of a class may allow different implementations. To a client of a class only the specification should be visible and not the concrete implementations. This prevents a client to take advantage of implementation decisions which are not directly related to the abstraction the class provides. The purpose of information hiding is to make clients independent from concrete implementations. Dependency on the implementation may cause a client's code to fail when the implementation is changed. Furthermore exposing the implementation may rather distract from the essence of the abstraction. Hence information hiding is an import concept to increase reliability of software and support reuse.

Information hiding implies that the language provides facilities to hide internal features used for the issue of implementation only. In *Java* modifiers like `private` or `public` are available. *Eiffel* allows a class to make features visible selectively to a designated set of classes. Visibility to any other class or none are the extremal cases of this so called *selective export*.

1.2.3 Inheritance

Information hiding separates the abstract specification cleanly from the concrete implementation. Different implementations of the same specification cannot be distinguished from outside. So information hiding does not allow to refine implementations. A refinement makes the behavior more specific and hence represents a new abstraction. However one that is compatible with the original abstraction, i.e. the original abstraction is preserved in the refinement. Thus it should be allowed to apply operations of the original abstraction to the refined one, too. The refined abstraction then defines a *subtype* of the more general data type. In object-orientation such a refinement is called *inheritance* – the properties of the general type are inherited by the refined type. One says that a class that inherits from another class *extends* this class. In this context the class which is subject of extension is called the *parent*, and the extended class is the *child* or *heir*.

On the abstract level inheritance means that the extended type provides at least the same interface as the parent and behaves accordingly. In object-oriented languages inheritance implies that the child not only inherits the behavior but also the implementations of the parent's features. But such an incremental extension is not always possible. As the extension may introduce new consistency constraints, the inherited implementation may not be suited to ensure these constraints. So it has to be possible to replace the inherited implementation by a new one.

From an abstract point of view whether an object takes the inherited implementation or provides a new one does not matter. It is the behavior that counts. Inheritance can be considered a procedure to copy implementations. However implementation inheritance is more than just a convenience to avoid reproduction of code. Information hiding prohibits to copy the implementation manually, as this information should not be accessible from outside. Implementation inheritance supports reuse in refinement without violation of information hiding.

Subtyping introduces a new facet of abstraction into programming. Algorithms or systems can be implemented on the most abstract level possible once and for all. What the exact nature of the manipulated objects is does not matter. The only requirement is that it fits the general abstraction. Hence subtyping allows to abstract from non-relevant details of the manipulated objects. But in this case the non-relevant details are not implementation aspects but are of an abstract nature. While information hiding is rather a technicality, subtyping enables abstraction in its literal sense.

In order to work correctly it has to be assured that for each object the appropriate implementation is executed. Thus the feature to be executed cannot be determined by a static program analysis, as this depends dynamically of the object's concrete type. The determination of the correct implementation at runtime is called *dynamic binding.* Therefore a mapping between the features of the declared type and the corresponding features of all subtypes has to be computed by the compiler.

It is quite natural that a refinement combines different abstractions to a new one. The resulting data type then is a subtype of all involved types. Extending more than one type is called *multiple inheritance. Eiffel* is one of the few object-oriented languages which allow multiple inheritance consequently. While *Java* only allow it on the level of interfaces, but not for classes.

1.3 Modular Verification

1.3.1 Modular Soundness

The modular nature of object-orientation suggest to do the verification also in a modular way, i.e. once and for all for each class. In order to achieve this we need the following property:

> *Modular Soundness* is the property that the separate verification of the individual modules of a program suffice to ensure the correctness of the composite program. [2]

Efficiency of reusability is hampered if modular verification is not possible. If the correctness of a reused class has to be verified anew in every program also the concept of information hiding may be violated. A client of a class should not depend on the implementation of that class. But exactly this is the case when the correctness of a reused class depends on the specific program context.

In modular proofs only the specifications of the reused classes may be used. Thus the specification becomes a critical part. It has to be concrete enough to express the behavior in all possible contexts – without having the possibility to foresee all this. This situation is also known as the *open world assumption*. If a specification turns out to be inadequate to deduce the properties a client needs to proof its own correctness, a specification cannot be changed – neither narrowed nor widened. Narrowing could turn a correct implementation incorrect, as it would introduce new assumptions to the abstraction which may not be met by a former correct implementation. Widening can be harmful as well, namely because the original specification has been used for proofs of other classes already. In both cases the correctness of other classes would be affected. So we have to concentrate on how abstract specifications can be stated in order to allow efficient reuse.

1.3.2 Specifications

An abstract specification of a class has to fulfill the following conditions:

1. It has to define the state of an object in an abstract way. This can be done by introducing so called *abstraction variables*, introduced by Hoare [1]. The values of such variables are in an abstract domain.
2. For every feature the effect on the object's state has to be defined. Also effects on the environment of the object have to be specified.
3. The former conditions allow to derive changes to the systems state. In an open world one also needs means to state in full generality which parts of the state remain unchanged.

Abstract States of Objects

In the object-oriented paradigm an object always represents a value of an abstract domain. We call the abstraction of a class also the *model* of the class. The object has an identity of its own and must not be confused with the value it represents. Thus the state of an object can always be expressed as the state of the corresponding model. For an abstract type some constraints may have to be satisfied to be in a consistent state. Such constraints are stated by so called (class-)*invariants*.

A correct implementation has to provide a mapping to the abstract model. As a dynamic data structure cannot be implemented by a single object the physical state of an object – the current assignment of its attributes – does not correspond directly to the abstract state. Moreover the correct translation of the invariants to the implementation may be difficult, since they may depend on the state of complex object structures.

Pre- and Postconditions

Properties that hold for each state of an object are expressed via invariants. In contrast to this a feature has a certain effect in execution which in general depends on the object's abstract state. In order to be applicable a feature may require some conditions to be satisfied. These requirements are usually stated in the so called *precondition*. After execution the feature ensures to leave the system a state characterized by the *postcondition*, provided that the precondition was fulfilled prior execution. Furthermore the feature has to guarantee that the class invariant is re-established.

In a correct program for every feature invocation both the caller and the callee have obligations. The caller is responsible to meet the feature's precondition, while the feature must ensure to satisfy the postcondition and preserve the class invariant. The pair of pre- and postcondition is often called the *contract* [7] of a feature, reflecting the situation of mutual obligations.

In the rest of this section we discuss how well such contracts are suited to specify the behavior of a class.

1.3.3 The Frame Problem

As pointed out above modular verification is primarily a problem of stating abstract specification concisely. I.e. complete enough to be reusable in a variety of contexts, but also in a form to be expressed as clear as possible. The latter condition addresses complexity of specification. If modular verification has to be bought with exploding complexity of the specifications, it might be doubted that efficient reuse is possible.

In general a program in an imperative language is composed of a sequence of instructions. Correctness proofs rely to an important part on the knowledge that certain properties are preserved when an instruction is executed. Thus the specification's ability to express what is preserved is crucial. This problem was first addressed by McCarthy and Hayes [5] in relation with artificial intelligence. The relevance of the frame problem in connection with modular soundness is also pointed out in the works of Leino [3], Leino and Nelson [2], Müller [11], and Müller and Poetzsch-Heffter [12].

1.3.4 Dependencies

Leino and Nelson [2] propose an approach to overcome the frame problem in modular verification and to achieve modular soundness. Besides pre- and postconditions they propose further specification constructs as *modifies lists*, *representation functions* and *abstraction dependencies*.

The initial idea of the approach is to declare explicitly in a *modifies list* which parts of the abstraction may be changed in a feature invocation.

In the abstract specification data is represented by abstract variables which are not manipulated directly by the program. In the compiled program these variables do not even exist. They are virtual constructs whose values depend on other variables, concrete or abstract. Thus an abstract variable is a function of one or more concrete variables. This function is called the *representation* of the abstract variable. The actual representations are defined by the implementations.

In a fixed context the modifies list can always be integrated into the pre- and postcondition of a feature. For every variable which is not modified by the feature this fact will be stated in the postcondition. As the contexts in which a feature will be invoked cannot be foreseen, the modifies list is needed to generate the corresponding assertions. The variables which are not modified can be regarded as the complement of the variables mentioned in the modifies list, with respect to all visible variables of the context. Thus modifies lists are more than 'syntactic sugar'.

Moreover what a feature is allowed to modify also depends on the context of invocation. An implementation of a feature that changes the state of an object must be allowed to modify the internal representation of the object, because the abstract state depends on the internal state. Due to information hiding the internal representation is not visible in the abstract context and hence the modifies list cannot mention it. The solution to this problem are *abstraction dependencies*. An abstract variable a is said to *depend* on a variable b (abstract or concrete), if a modification of b may change the value of a. The license to modify a variable implies also the license to modify the variables it depends on. By this the implementation can accomplish its task and modify the variables on which the abstract state depends on. Leino and Nelson point out that an explicit declaration of what the modified variables depend on is necessary, since this information cannot be inferred automatically. The dependencies allow to complete the modifies list so that all variables which may be modified by a feature call occur.

Leino and Nelson distinguish two kinds of dependencies, namely static and dynamic ones. While the former dependency can be determined statically the latter involves the dynamic state of the system. For static dependencies Leino and Nelson have proved modular soundness of their approach, while they could not yet proof this for dynamic dependencies.

1.3.5 An Alternative Approach

Bertrand Meyer [9] provides an alternative approach to class verification. Basic programming constructs as e.g. sequences of instructions or the famous 'dot'-operator are defined in terms of operators on mathematical functions. This implies that every language construct can be interpreted as a mathematical function, thus one needs a denotational style semantics. Also specifications, in form of a pair of a pre- and postcondition are to be integrated in this mathematical approach. Verification of a feature, equipped with a specification becomes to an evaluation of a boolean function. Only if this evaluation yields the constant function 'true' the implementation is correct.

The evaluation of a feature evaluation is based on rules like associativity and distributivity which hold for operators. Transformation of the program is done according those laws.

It is to say this approach is rather in an early stage, compared with the approaches discussed above. Subtyping and side-effects cannot be handled yet. Moreover there is no concept available how to deal with feature calls in an abstract way. So it is to expect that the problem of abstract specifications also will be encountered in this approach.

1.4 Outlook

This introduction gave a brief overview on program correctness and object-orientation. The major problems of modular verification were presented. In the rest of this report we will see an example language inspired by *Eiffel*, together with a formal semantics. This language is restricted to some central object-oriented concepts, presented in the next chapter. The aim of this language was to provide a language in that is well suited for modular correctness proofs. It has to be stated clearly that the order in which the results are presented in this report does not follow the chronology of the project. In the project the starting point was to define the semantic definition of an object-oriented language. During this process some limitations of the contracts in *Eiffel* became apparent. Due to these limitations the focus was led to treatments of the frame problem and the approaches of Leino, Nelson, Müller and Poetzsch-Heffter. Hence one may not expect the language presented here to solve any of the problems discussed in this introduction. The result of this work rather can be considered to emphasize the importance of these problems with respect to modular program verification. The last chapter will readopt these considerations.

Chapter 2

Example Language

The most important property of a program is correctness. Of course correctness is not an absolute property. A program is not correct or incorrect per se, but only with respect to a specification. If a program is not correct it cannot have much other useful properties, as e.g. efficiency. This immediately raises the question whether a program without specification is of much use? It does not have to be completely useless. Possibly it was written with a clear idea what its purpose should be, but unfortunately the program lost this specification somehow. This is a typical case of poor documentation, making efficient reuse of existing software impossible, as one has to figure out tediously what the program's purpose might be. But in much cases the absence of a specification is just due to ignorance about the problem to be solved.

In this light specification not only enables one to ensure correctness of a program, it also may help to get a deeper insight into the problem to be solved. Thus the labor on a good specification may also have a positive impact on the final software solution.

A serious problem in programming is the conceptual gap between specification and implementation. There are few programming languages which support means to describe program specifications within the language. So it is often the case that the meaning of a specification cannot be mapped directly to the level of the programming language.

One of the aims of this project is to investigate possibilities to bring the specifications – or at least parts of them – into the program itself. Thus the notion of correctness could be defined formally using the formal semantic definition.

A good starting for these investigations may be the *Eiffel* programming language [8]. It is designed not only for implementing the final solution, but also to support the whole software construction process. From the design and specification stage to the final product. An important part of the language are so called *assertions*. They comprehend class and loop invariants, pre- and postconditions and checks which can be placed in the program. These assertions have two purposes. On one hand they are part of the specification and also documentation. On the other hand they can be checked at runtime during the testing phase. Thus errors can be traced easily to its origin, namely the assertion which was violated.

The language to be devised here is strongly influenced by *Eiffel*. Apart from some details it can be considered a subset of the *Eiffel* language. The description follows a top down approach, describing briefly the concepts which will be supported. The syntax of the language will be described informally. In some cases we simply refer to *Eiffel*.

2.1 Classes

The class is the elementary unit from which a program is built. In fact a program is a finite collection of classes which are interacting. In order to specify the kinds of interactions the class declaration has to indicate to which types it is conformant, the functionality it provides to client classes and how instances of the class can be obtained if the class is effective.

2.1.1 Types

Every class defines a type in the system. In order to determine the type hierarchy a class has to indicate its relative place in the type system, i.e. it has to declare from which classes it does inherit directly. This information is exposed in the inheritance clause of the class declaration.

Inheritance obligates the class to implement the inherited features appropriately. Furthermore the compiler has to be able to determine the feature to be called in relation with dynamic binding. For our purposes it suffices that a mechanism exist to determine correspondence between the features of some type to its subtypes. For the inheritance clause we borrow the mechanism provided by the *Eiffel* language [8].

2.1.2 Creation

A class is effective if it can have instances at runtime. In order to represent a consistent abstraction an object has to be initialized accordingly. We require this initialization to be explicitly declared. The creation-clause lists the commands that can be used for initialization of an object. The precondition of a creation feature must not rely on any property of the object to be initialized. Since deferred classes cannot have instances they must not have a creation clause.

2.1.3 Feature List

In the features section of a class declaration all (publicly available) features have to be listed. Furthermore for implementation issues some helper features may be introduced which are not visible to clients. The public part of the feature list can be regarded to be the interface of the class. The public part of the feature list is indicated by 'ANY', the private part by 'NONE'.

Eiffel allows to export features selectively, i.e. a set of types may be declared to which a certain feature is visible. We do not allow such graduations of visibility, as this implies that a class may represent different abstractions to different clients. The arising complexity should be avoided here.

2.1.4 Invariant

The invariant clause is part of the specification of the class. It communicates properties a client always can rely on. Therefore a correct implementation of a class has to ensure that the invariant is re-established after every feature invocation.

2.1.5 Informal Syntax

```
class Name
  inherits
    InheritsDeclaration
  creation
    CreationList
  feature {ANY}
    FeatureList
  feature {NONE}
    FeatureList
  invariant
    InvariantDeclaration
end
```

2.2 Boolean and Integer Objects

Up to now we considered only objects which are able to represents any object of the abstraction domain. For some abstraction domains, like boolean values or the integers, this approach does not seem suitable. It does not make much sense to speak from different copies of the value 'True' or the number 0. To represent these values as ordinary objects which can be allocated is rather an abstraction of a memory cell than of the value. So when two objects represent the value 'True' then we want these objects to be equal.

There are two approaches to face this problem. One can introduce a category of value types which are not considered objects. This approach is taken in Java where boolean values and numbers are so called base types. The consequence is that they have to be used differently than objects. Therefore these types are modeled as classes all the same in order to use them in the role of objects.

The second approach does not treat these types apparently different than normal classes. But the instances of the class rather *are* the objects of the abstraction domain than representatives. Thus these objects have to be stateless. Moreover they do exist without prior creation, as any abstract object does.

Therefore we call them also 'a priori' objects. Since these objects cannot be created they have to be accessible by name.

In our language we follow the latter approach. The predefined types we support are:

- $\mathbb{B} = \{\text{True, False}\}$
- $\text{INTEGER} = \{\ldots, -1, 0, 1, \ldots\}$
- $\text{NONE} = \{\text{Void}\}$

2.3 Feature Declaration

The feature declaration first describes its abstract behavior. Optionally, if the class should be effective, it also may provide an implementation. The abstract behavior is described by the pre- and postcondition of a feature.

Features may be of one of two kinds: Either a query returning a result or a command resulting in a state change. The purpose of a query is to compute a value or to query properties of the current state. Ideally a query has no side-effects, i.e. a query invocation should not change the state. However one can interpret the state in different ways. For a client side-effect freeness is already achieved if the state of its abstract view of the system is not changed. On the lower level of the system state the query may have a side-effect. An implementation of a date structure may e.g. adapts its internal structure according to the frequency of certain queries. This restructuring may increase the performance of the data structure, but it does not change the abstract state. Thus to forbid side-effects completely for queries may be to restrictive. Moreover the proof of side-effect freeness may be non-trivial.

Queries can be implemented in two ways. Either as attributes or routines. An attribute is a reference to an object. Thus the query result is stored rather than computed. Obviously an attribute does not take arguments. Attributes may be changed during execution, so the internal state of an object is determined by the values the attributes refer to.

The decision whether a query is implemented as an attribute should not be visible to the clients. This principle is called *uniform access*. Eiffel enforces this principle, and we also do. Opposite to this in Java the distinction between attributes (there called fields) and routines (methods) is visible in every call. Furthermore a visible field can be modified by a client. This may violate the consistency of the object. Therefore it is reasonable to allow only the owning object to modify its attributes.

An effective class has to provide an implementation for each feature. We distinguish feature declaration and implementation as different constructs which will have different semantics. The set of feature declarations in a class is denoted by 'Features', while 'FeatureImpl' contains all implementations. The declaration represents a client's view on a feature, i.e. its abstract specification. With dynamic binding a feature invocation may be redirected to another implementation than that of the class declared. In the case of deferred classes the invocation always has to be redirected.

In a feature implementation local variables can be declared. A local variable is a reference to an object. Variables are not visible from outside, and hence are not subject of side-effects.

Attribute

Name : *Type*

Query Routine

```
Name Arguments : Type is
  require Precondition
  local Variables
  do
    RoutineBody
  ensure Postcondition
end
```

Command Routine

```
Name Arguments is
  require Precondition
  local Variables
  do
    RoutineBody
  ensure Postcondition
end
```

2.4 Routine Body

The syntactic constructs presented so far are rather concerned with structuring the environment. They declare which types and features are available and how the type hierarchy is constructed. But they do not directly describe how the state of the system is manipulated. Opposite to this the constructs used to implement a routine primarily describe state transformations.

2.4.1 Instruction

Instructions are the basic construct to change the state of a system. They are the elementary unit to build complex blocks of functionality.

Instruction ::= *Assignment* | *Conditional* | *Loop* | *Command* | *Creation*

Instruction Sequence

Instructions can be executed in sequence. The sequencing operator is denoted by a semicolon ';'. The body of a routine implementation in general is an instruction sequence. In the trivial case an instruction sequence consists of one instruction only.

InstrSeq ::= *Instruction* [**;** *InstrSeq*]

Assignment

With an assignment one can change the value of local variable or of an attribute of the current object.

Assignment ::= *Local* := *Expression* | *Attribute* := *Expression*

For the definition of the attribute see section 2.4.2.

Local and Attribute

The production *Local* describes a query on a local variable which is declared in the routine. It is the compiler's obligation to ensure that only declared variables can be used. A local variable is dereferenced by simply writing its name into the program code.

Local ::= *Name*

The production *Attribute* describes an attribute of the current object.

Attribute ::= *Name*

Conditional

A conditional instruction allows to choose between two alternative instruction sequences, depending on a condition. If evaluation of the condition yields 'True' the instruction sequence in the *then*-part is chosen, otherwise the sequence in the *else*-part.

Conditional ::= **if** *BoolExp* **then** *InstrSeq* **else** *InstrSeq* **end**

Loop

Using a loop some instruction sequence can be executed several times, until some termination condition is satisfied. Before the loop body is executed the first time an initialization part is executed and the termination condition is evaluated for the first time.

Loop ::= **from** *InstrSeq* **until** *BoolExp* **loop** *InstrSeq* **end**

Command

A command is either the invocation of a command feature on the result of an expression or on the current object.

Command ::= *Expression*.*CommandInvocation* | *CommandInvocation*

CommandInvocation

A command invocation simply denotes the invocation of a command feature.

CommandInvocation ::= *FeatureInvocation*

Creation

A creation instruction allocates a new object which is initialized by the indicated creation feature. Then the newly allocated object is assigned to the local variable or attribute indicated.

Creation ::= **create** *Local*.*CommandInvocation*
| **create** *Attribute*.*CommandInvocation*

2.4.2 Expression

In contrast to an instruction an expression returns an object. Its main purpose is to perform computations or to query the state. Expressions may have side-effects in general.

Expression ::= *Local*
| *Expression*.*Query*
| *Query*
| (*EqTest*)
| **Current**

Dot-Operator

Given a reference to an object a feature of this object is accessed via the so called *dot-operator*. So it is possible to invoke a query on the result of an expression and build chains of query invocations.

Query

A query is an invocation of a query feature.

Query ::= FeatureInvocation

Equality Test

An equality test is an expression which returns a boolean value, depending on whether the compared objects are the same or not.

$EqTest ::= Expression = Expression \mid Expression \neq Expression$

Current

The keyword 'Current' is used in a routine body to refer to the target relative to which the feature is executed. A reference to the current object is needed e.g. when it has to be passed as argument to a feature invocation.

Boolean Expressions

A boolean expression is an expressions with the constraint that it returns a boolean object.

$BoolExp ::= Expression$

2.4.3 Other Constructs

FeatureInvocation

A feature is invoked by indicating its name. If the feature takes arguments these are passed using a tuple.

$FeatureInvocation ::= Name \mid Name\ Tuple$

Tuple

A tuple is a finite list of expressions enclosed in parentheses. Empty tuples are not allowed.

$Tuple ::= (Expression\ \{,Expression\})$

Chapter 3

Operational Semantics

In this chapter an operational semantics is provided for the language sketched in chapter 2. The semantic definition has to describe the effect of each syntactical construct to the abstract state of the system. The main reason for choosing an operational style semantics is that the definition of recursive structures like loops and functions can be done straight forward, without using fixed point theory. Furthermore an operational semantics may server as the basis to develop a denotational or axiomatic semantics.

3.1 Static Environment

The static environment consists of mathematical objects which are independent of a concrete program. They belong to the part of the mathematical model which cannot be influenced by a programmer. These objects form the basis on which the whole semantical model is built on.

3.1.1 Objects

In an execution of a program objects – in the object-oriented sense – are manipulated. These objects all belong to the domain 'Objects'. The number of objects that can be allocated during program execution is not bounded. Therefore an infinite reservoir of objects is needed. However during computation not all elements of 'Objects' will be allocated. In fact in every state only a finite number of objects is allocated. So one can say that 'Objects' represent rather the set of *potential objects* than the set of objects actually used.

The set 'Objects' also contain the predefined objects of types boolean and integer. In section 2.2 we decided that they are stateless and their identity coincides with the value they represent. These objects cannot be allocated because they are always accessible by name. So these objects have to be distinguishable from other objects. This distinction is done by the type system. As will be explained below (section 3.3) every effective class has its own reservoir of instances. The predefined types are considered to be effective, and hence their instances are separated from instances of other types.

3.1.2 Names

In a program classes, features and local variables can be declared. In order to refer to these declared entities names have to be given to them. The set 'Names' consists of all valid identifiers which can be used as a name.

How names are formed is not of interest in the semantic definition. It is the compiler's obligation to ensure that all names in a program are valid – this includes especially non-ambiguity.

The 'a priori'-objects are addressed by name, so there are some names reserved for these objects. The most prominent are 'True', 'False' and 'Void'.

In query-routines the name 'Result' is reserved for the local variable which will store the result to be returned by the query. Thus a programmer is not allowed to declare a local variable with the same name in a query-routine.

3.2 State

3.2.1 Abstract or Explicit?

In much semantic definitions the state of the system is treated as a black box from which information can be extracted. I.e. one abstracts from the representation of the state. At first sight this approach promises to be easily extendable. Introducing new aspects of the state is done by defining appropriate extraction functions. However this abstract approach does not provide this freedom at all. There are two kinds of state transformations to distinguish. The first kind are those which can be defined completely in terms of other transformations. In this case it does not matter what the state is. Thus the explicit and abstract approach are equivalent in this case. The second kind are transformations which only can be defined by constituting constraints on the result state. Adding a new aspect to a state usually also means to refine such constraints for the new aspect appropriately. Thus incremental extendability is not gained with the abstract approach.

In the end the aspects introduced to the state define a view on the state which is not different to an explicit definition. There may be some hidden structures in the state not present in this view. Thus a view on an abstract state in fact is an equivalence relation on the set of states. But when we do not have any knowledge about those structures, why should we care about them? Therefore the state definition in the following treatise is explicit.

3.2.2 State Components

What parts of the system should belong to the state? In the beginning it is tempting to model the state in analogy to an existing computer environment, although in an abstract way. This approach may be suited when one aims to build a compiler. However for a semantics that operates primarily on the source code of a single routine a concept as the execution stack is not of interest. The same holds for the current object – within a routine the current object does not change.

We define the state to consist of three parts: The assignment of the local variables, the assignment of the attributes of each object and a relation on 'Objects' which defines the set of allocated objects.

$$\text{Locals} := \text{Names} \hookrightarrow \text{Objects} \tag{3.1}$$

$$\text{Attributes} := \text{Objects} \to \text{Names} \hookrightarrow \text{Objects} \tag{3.2}$$

$$\text{Allocated} := \text{Objects} \to \mathbb{B} \tag{3.3}$$

The functions are indicated to be partial because the number of local variables and attributes is always finite. Hence the assignments cannot be total.

The state is defined as the Cartesian product of these three function sets.

$$\text{States} := \text{Locals} \times \text{Attributes} \times \text{Allocated} \tag{3.4}$$

The components of a state are obtained by extraction functions:

$$\text{locals} : \text{States} \to \text{Locals} \tag{3.5}$$

$$\text{attributes} : \text{States} \to \text{Attributes} \tag{3.6}$$

$$\text{allocated} : \text{States} \to \text{Allocated} \tag{3.7}$$

3.2.3 Constraints

The 'allocated' component does not rightly fit the chosen abstraction level. As a programmer does not care for the execution stack, he is not interested in the non-allocated objects. Every object reachable from an allocated object is expected to be an allocated object of the state as well. A property which unfortunately is not guaranteed by every element of 'States'.

The reason why the 'allocated'-component is needed is object creation. There it has to be guaranteed that a certain feature is invoked relative to an object which is not in use already. This is also the only case, where the 'allocated' component of the pre- and poststate differ.

A state σ is called *consistent* if it has the following properties:

$$\begin{array}{l} \forall n \in \text{Names}\ \forall u \in \text{Objects}\ (\text{locals}\ \sigma\ n = u \Rightarrow \text{allocated}\ u) \\ \forall n \in \text{Names}\ \forall w, u \in \text{Objects}\ (\text{attributes}\ \sigma\ w\ n = u \Rightarrow \text{allocated}\ u) \end{array} \tag{3.8}$$

This constraint imposes the obligation to proof that all transformations preserve state consistency. What a transformation does to inconsistent states is not of interest.

To model the set of allocated objects in this way prohibits to reason about automatic memory reclamation (*garbage collection*). Whether an object may ever be reachable in the future computation cannot be decided when the execution stack of all feature invocations which have not yet terminated is not known. Therefore allocation of new objects is done in a conservative way, assuming that the reservoir of potential objects is infinite.

3.2.4 Semantic Functions

A semantic definition has to define a state transition function for each construct which may change the state – this semantics is called the *transformation semantics*. Additionally some constructs have a result value. The state, as it is defined above, does not allow to contain a result value as part of the state. Therefore every construct is also supposed to have a *query semantics*, defining the result value.

For a given syntactic construct s the corresponding query and transformation semantics are denoted by $[\![s]\!]_{\mathcal{Q}}$ and $[\![s]\!]_{\mathcal{S}}$ respectively. The final result of the former is an object, while the latter yields a state. The notation $[\![s]\!] := ([\![s]\!]_{\mathcal{Q}}, [\![s]\!]_{\mathcal{S}})$ stands for the *overall semantics* of s.

In combination with invariants and pre- and postconditions the query semantics enables one to formulate what it means for a program to fulfill its contract. An assertion is satisfied if its query semantics yields the 'True'-object.

Side-Effect-Freeness of Assertions

Any kind of assertion intends to introduce information in the program code which is related to program correctness, and is part of the specification. If a program meets its specification, the effect on the state ideally should be the same as if the assertions were not present. This implies that checking of assertions is supposed to leave the state the same if the check evaluates to true. If only provably correct programs are allowed to be executed, the question of what to do if an assertion is violated is not raised. Therefore assertions are supposed to have no effect on the state. However this side-effect-freeness cannot be guaranteed by the compiler. So we leave it to the responsibility of the programmer not to write assertions with side-effects.

3.3 Classes

3.3.1 Relationship to 'Objects'

A program consists of a set of class declarations, denoted by 'Classes'. A class defines a set of instances which are members of 'Objects'. Thus a class C can be considered a subset of 'Objects'.

$$C \subset \text{Objects} \tag{3.9}$$

Every class also defines a type. Since subtyping is possible, a type can comprehend the instances of different classes. But an object can only be an instance of a single class. On the other hand any object is supposed to be an instance of some class. Thus the classes establish a partition on the set 'Objects'.

If an object u is an instance of class C, we say that the *actual type* of u is C. The actual type is most specific in the sense that u belongs not to a proper subtype of its actual type. One can show that the most specific type of an object is unique – i.e. the most specific type of an object is always the actual type.

The partition property of 'Objects' enables us to define a function 'instance' that determines the actual type of an object.

$$\begin{aligned}&\text{instance} : \text{Objects} \to \text{Classes}\\ &\text{instance } u = C \;\; :\Leftrightarrow \;\; u \in C\end{aligned} \tag{3.10}$$

3.3.2 Features and Dynamic Binding

In a class each feature defines a semantic function. The signatures of these functions depend on the number of arguments the feature takes. How the semantics of a feature implementation is obtained from the program code is described in section 3.4.

Features of a class may be exported for use by instances of other classes, or they may be private, i.e. only to be invoked from the current object. As subtyping is allowed, an invocation of some exported feature f on an object of static type t is redirected to the corresponding feature f' of the objects actual type t'. Since the class structure of an application is fixed, this redirecting mechanism can be defined in advance and so does not depend on the state. The redirecting is known as *dynamic binding*. The corresponding function to determine the target of the binding is denoted by 'bind'.

$$\text{bind} : \text{Classes} \to \text{Classes} \hookrightarrow \text{FeatureImpl} \tag{3.11}$$

Where 'FeatureImpl' consists of all feature implementations.

From the programmer's point of view a feature is not bound on some specific implementation. So we distinguish between the abstract feature and its implementations. The abstract feature comprehends all possible implementations in the subtypes. Therefore the semantics of a feature is primarily concerned with dynamic binding.

The set 'Features' contains all abstract features, while 'FeatureImpl' contains the concrete feature implementations, which are the target of dynamic binding. The signatures of the semantic functions for a feature and its implementations are the same.

Let f be a feature and u an object.

$$\begin{aligned}&[\![f]\!]_{\mathcal{Q}}\ u := [\![(\text{bind } C \ (\text{instance } u))]\!]_{\mathcal{Q}}\\ &[\![f]\!]_{\mathcal{S}}\ u := [\![(\text{bind } C \ (\text{instance } u))]\!]_{\mathcal{S}}\end{aligned} \tag{3.12}$$

The consistency property (3.8) on states is satisfied if the targets of the binding do satisfy it.

3.4 Feature Implementations

In section 3.3.2 the abstract semantics of a feature was defined. The abstract semantics comprehends the semantics of all features to which the invocation may be redirected.

3.4.1 General Remarks

The semantic functions of a feature implementation f may have one of the following forms:

1. If the feature does not take arguments:

$$\begin{aligned}&[\![f]\!]_{\mathcal{Q}} : \text{Objects} \to \text{Objects} \to \text{States} \to \text{Objects}\\&[\![f]\!]_{\mathcal{S}} : \text{Objects} \to \text{Objects} \to \text{States} \to \text{States}\end{aligned}$$

2. If the feature takes $n > 0$ arguments:

$$\begin{aligned}&[\![f]\!]_{\mathcal{Q}} : \text{Objects}^n \to \text{Objects} \to \text{Objects} \to \text{States} \to \text{Objects}\\&[\![f]\!]_{\mathcal{S}} : \text{Objects}^n \to \text{Objects} \to \text{Objects} \to \text{States} \to \text{States}\end{aligned}$$

The semantics of a feature implementation depends on the passed arguments (provided the feature takes arguments), the object in whose context the feature is invoked, the target object of the invocation and the state. The object of the context of invocation is used when feature invocations are composed to chains by the dot-operator. Since arguments in such chains are always evaluated relative to this context, it has to be passed to subsequent invocations (see also section 3.5.9).

3.4.2 Attributes

Attributes of a class have a rather simple semantics. An invocation has no side-effects. The result of an invocation is the value assigned to this attribute with respect to the current object.

Let a be an attribute of the current class. The query- and transformation-semantics are defined as follows:

$$\begin{aligned}&[\![a]\!]_{\mathcal{Q}} : \text{Objects} \to \text{Objects} \to \text{States} \to \text{Objects}\\&[\![a]\!]_{\mathcal{Q}}\ u\ v\ \sigma := \text{attribute}\ \sigma\ v\\&[\![a]\!]_{\mathcal{S}} : \text{Objects} \to \text{Objects} \to \text{States} \to \text{States}\\&[\![a]\!]_{\mathcal{S}}\ u\ v\ \sigma := \sigma\end{aligned} \tag{3.13}$$

Since an attribute invocation does not have any effect on the state, consistency according to (3.8) is preserved.

3.4.3 Routines

In contrast to an attribute a routine has an implementation which cannot be changed during computation. The semantic effect of a routine is defined by the routine body. Execution of a routine cannot start in an arbitrary state. A valid state σ has to fulfill the following three conditions:

1. For every declared local variable v of type t the function application

$$\text{locals}\ \sigma\ v$$

must yield the default value of the corresponding type t. In the case of an anonymous type, the default value is 'Void', while for an 'a priori'-type the default value is specified in the type definition for t.

2. For every formal argument p (if any) of type t the function application

   $$\text{locals } \sigma \ p$$

   yields the object passed by the caller.

3. For any name n which denotes neither a local variable nor a formal parameter the function application

   $$\text{locals } \sigma \ n$$

   is not defined.

When assertion checking is active additional steps are executed (see 3.6).

There are no constraints on the 'attribute' component of the state σ. This part is taken as is from the caller. For every routine a designated initialization function – denoted by 'init' – defines the initial state, depending on passed arguments and state. The following property states that the object-structure is not changed by 'init':

$$\text{attributes } (\text{init } \sigma) = \text{attributes } \sigma \tag{3.14}$$

Furthermore after execution of the body the formerly passed 'locals' component must be restored, since function invocations should not change the local variables of the caller. The corresponding function is denoted by 'restore'. It takes two states and returns a state consisting of the local-component of the first argument and the 'attributes' and 'allocated' component of the second one. Thus the effects on the object structure become visible to the caller.

$$\begin{aligned} &\text{restore} : \text{States} \to \text{States} \to \text{States} \\ &\text{restore } \sigma \ \sigma' := (\text{locals } \sigma, \text{attributes } \sigma', \text{allocated } \sigma') \end{aligned} \tag{3.15}$$

Routines without Arguments

Let r be a routine without arguments. The corresponding semantic functions have the following signatures:

$$\begin{aligned} &[\![r]\!]_{\mathcal{Q}} : \text{Objects} \to \text{Objects} \to \text{States} \to \text{Objects} \\ &[\![r]\!]_{\mathcal{S}} : \text{Objects} \to \text{Objects} \to \text{States} \to \text{States} \end{aligned} \tag{3.16}$$

The initialization function transforms a passed state σ to a state σ' for which locals σ' satisfies the above constraints. The 'attributes' component is not changed by this transformation – i.e. the invoked function manipulates the object structure passed by the caller.

Let s be the body of routine r. Syntactically the body is an instruction sequence (see section 3.5.3). Query-routines return a result value, stored in the local variable 'Result', while commands have no result. Therefore we have to distinguish queries from commands for the query semantics. For a query r the semantic functions are defined as follows:

$$\begin{aligned}&[\![r]\!]_{\mathcal{Q}}\ u\ v\ \sigma := \text{locals}\ ([\![s]\!]_{\mathcal{Q}}\ v\ (\text{init}\ \sigma))\ \text{Result}\\&[\![r]\!]_{\mathcal{S}}\ u\ v\ \sigma := \text{restore}\ \sigma\ ([\![s]\!]_{\mathcal{S}}\ v\ (\text{init}\ \sigma))\end{aligned}\tag{3.17}$$

If r is a command, the definitions are:

$$\begin{aligned}&[\![r]\!]_{\mathcal{Q}}\ u\ v\ \sigma := \text{undefined}\\&[\![r]\!]_{\mathcal{S}}\ u\ v\ \sigma := \text{restore}\ \sigma\ ([\![s]\!]_{\mathcal{S}}\ v\ (\text{init}\ \sigma))\end{aligned}\tag{3.18}$$

Routines with Arguments

Let r be a routine with n arguments. The signatures of the corresponding semantic functions are as follows:

$$\begin{aligned}&[\![r]\!]_{\mathcal{Q}} : \text{Objects}^n \to \text{Objects} \to \text{Objects} \to \text{States} \to \text{Objects}\\&[\![r]\!]_{\mathcal{S}} : \text{Objects}^n \to \text{Objects} \to \text{Objects} \to \text{States} \to \text{States}\end{aligned}\tag{3.19}$$

The initialization function depends on both, the passed arguments and the passed state. Again, only the 'locals' component of the state is changed, according to the constraints stated above. Let s be the body of routine r. The semantic functions of a query r are defined as follows:

$$\begin{aligned}&[\![r]\!]_{\mathcal{Q}}\ p\ u\ v\ \sigma := \text{locals}\ ([\![s]\!]_{\mathcal{Q}}\ v\ (\text{init}\ p\ \sigma))\ \text{Result}\\&[\![r]\!]_{\mathcal{S}}\ p\ u\ v\ \sigma := \text{restore}\ \sigma\ ([\![s]\!]_{\mathcal{S}}\ v\ (\text{init}\ p\ \sigma))\end{aligned}\tag{3.20}$$

While for a command r the definitions are:

$$\begin{aligned}&[\![r]\!]_{\mathcal{Q}}\ p\ u\ v\ \sigma := \text{undefined}\\&[\![r]\!]_{\mathcal{S}}\ p\ u\ v\ \sigma := \text{restore}\ \sigma\ ([\![s]\!]_{\mathcal{S}}\ v\ (\text{init}\ p\ \sigma))\end{aligned}\tag{3.21}$$

3.4.4 Local Variables

The local variables declared in a feature define semantic functions available in the routine's body. A query on a local variable has no side effects at all. The value is completely defined by the 'locals' component of the state. Let v be a local variable.

$$\begin{aligned}&[\![v]\!]_{\mathcal{Q}} : \text{Objects} \to \text{States} \to \text{Objects}\\&[\![v]\!]_{\mathcal{Q}}\ u\ \sigma := \text{locals}\ \sigma\ v\\&[\![v]\!]_{\mathcal{S}} : \text{Objects} \to \text{States} \to \text{States}\\&[\![v]\!]_{\mathcal{S}}\ u\ \sigma := \sigma\end{aligned}\tag{3.22}$$

As one may notice, the parameter u of type Objects is not used in the definitions of the semantic functions. It is present all the same, because it may have to be passed to a subsequent feature invocation of an invocation chain (see 3.5.9).

3.5 Routine Body

The routine body contains the concrete implementation of a routine. In such an implementation constructs like local variables, attributes and features are used to achieve the intended effect on the state. The syntactic constructs discussed primarily have the purpose to combine the constructs treated in the previous sections to more complex structures.

3.5.1 Instruction

The instruction is the most elementary syntactic construct in a routine body. Its effect dependent on an object and a state. An instruction only causes a state transition but does not return a value. Here we only declare the signature of the semantic functions. The effect for each type of instruction is defined below. Let i be an instruction:

$$\begin{aligned}&[\![i]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\&[\![i]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\end{aligned} \tag{3.23}$$

3.5.2 Expression

An expression is used to compute a value. This value depends on a target object and a state. It returns a result and also may change the state. Let e be an expression:

$$\begin{aligned}&[\![e]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\&[\![e]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\end{aligned} \tag{3.24}$$

3.5.3 Instruction Sequences

To write instructions in sequence is the most important mechanism to build programs from elementary pieces. The characteristics of subsequent execution are:

1. Every instruction of the sequence is executed relative to the same object, namely the *current* object.

2. Every instruction operates on the result state of its direct predecessor, if present. The first instruction of a sequence starts with the initial state.

3. The semantics of an instruction in a sequence is independent of the query-semantics of its predecessors.

4. The semantics of an instruction sequence has the same signature as the semantics of an instruction.

The following definition reflects these properties for given instructions s and t.

$$\begin{aligned}
&[\![s;t]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\
&[\![s;t]\!]_{\mathcal{Q}}\ u\ \sigma := [\![t]\!]_{\mathcal{Q}}\ u\ ([\![s]\!]_{\mathcal{S}}\ u\ \sigma)\\
&[\![s;t]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\\
&[\![s;t]\!]_{\mathcal{S}}\ u\ \sigma := [\![t]\!]_{\mathcal{S}}\ u\ ([\![s]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.25}$$

3.5.4 Assignment

By an assignment the value of a local variable or an attribute of the current object is changed. Since local variables and attributes are represented differently in the state, the effect is different for these two kinds of assignment.

Assignment to Local Variable

Let v be a local variable and q be an expression. An assignment $v := q$ causes a state transformation to a state where the variable v has the value of q in the prestate. An assignment is an instruction and therefore has no query semantics.

$$\begin{aligned}
&[\![v := q]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\
&[\![v := q]\!]_{\mathcal{S}} = \text{undefined}
\end{aligned} \tag{3.26}$$

$$\begin{aligned}
&[\![v := q]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\\
&[\![v := q]\!]_{\mathcal{S}}\ u\ \sigma = \sigma',\ \text{where}\\
&\ \text{locals}\ \sigma'\ x = \begin{cases} [\![q]\!]_{\mathcal{Q}}\ u\ \sigma, & \text{if } x = v\\ \text{locals}\ \sigma\ x, & \text{otherwise}\end{cases}\\
&\ \text{attributes}\ \sigma' = \text{attribute}\ ([\![q]\!]_{\mathcal{S}}\ u\ \sigma)\\
&\ \text{allocated}\ \sigma' = \text{allocated}\ ([\![q]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.27}$$

Assignment to Attribute

Let a be an attribute and q be an expression. An assignment to a does not change the 'local' component of a state. After the assignment a has the value of q in the prestate.

$$\begin{aligned}
&[\![a := q]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\
&[\![a := q]\!]_{\mathcal{Q}} = \text{undefined}
\end{aligned} \tag{3.28}$$

$$\begin{aligned}
&[\![a := q]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\\
&[\![a := q]\!]_{\mathcal{S}}\ u\ \sigma = \sigma',\ \text{where}\\
&\ \text{locals}\ \sigma' = \text{local}\ \sigma\\
&\ \text{attributes}\ \sigma'\ w\ n = \begin{cases} [\![q]\!]_{\mathcal{Q}}\ u\ \sigma, \ \text{if } n = a\ \wedge\ w = u\\ \text{attributes}\ ([\![q]\!]_{\mathcal{S}}\ u\ \sigma)\ w\ n, \ \text{otherwise}\end{cases}\\
&\ \text{allocated}\ \sigma' = \text{allocated}\ ([\![q]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.29}$$

3.5.5 Creation

A creation instruction assigns to a local variable or an attribute a new instance of the declared type. In order to guarantee that the object will be in a consistent state a designated creation feature is invoked. So creation involves three steps:

1. Allocate an instance of the correct type. By this the state, namely the 'allocated' component, is changed to contain the newly allocated object.
2. Invoke the creation feature relative to the new instance.
3. Assign the created object to the variable or attribute to which the creation instruction is related.

From the programmer's point of view all non-allocated objects of a class are equivalent. So which of them is actually allocated should not influence the intended effect of a program. Therefore we do not define a concrete allocation strategy, but rather assume that an allocation function is defined in the environment which allocates objects correctly.

Object Allocation

In analogy to the semantics functions allocation is defined by a pair of functions $\text{alloc}_{\mathcal{Q}}$ and $\text{alloc}_{\mathcal{S}}$. The former returns the allocated object while the latter describes the transformation to the state.

$$\begin{aligned}&\text{alloc}_{\mathcal{Q}} : \text{Classes} \to \text{States} \to \text{Objects}\\&\text{alloc}_{\mathcal{S}} : \text{Classes} \to \text{States} \to \text{States}\end{aligned} \tag{3.30}$$

The next property states that the new instance is not allocated in the state relative to which the allocation is invoked.

$$\text{alloc}_{\mathcal{Q}}\ c\ \sigma = u \ \Rightarrow\ (\text{instance}\ u = c) \wedge (\neg\text{allocated}\ \sigma\ u) \tag{3.31}$$

In the poststate the new object will be allocated, but nothing else should be changed.

$$\begin{aligned}&\text{alloc}_{\mathcal{S}}\ c\ \sigma = \sigma',\ \text{where}\\&\quad \text{locals}\ \sigma' = \text{locals}\ \sigma\\&\quad \text{attributes}\ \sigma' = \text{attributes}\ \sigma\\&\quad \text{allocated}\ \sigma'\ v = ((\text{alloc}_{\mathcal{Q}}\ c\ \sigma = v) \vee \text{allocated}\ \sigma\ v)\end{aligned} \tag{3.32}$$

Semantics of the Creation Instruction

Let x be a local variable or attribute of declared type c, and let m be a command invocation (i.e. a feature). Then the effect of the creation instruction is defined as follows:

$$\begin{aligned}&[\![\textbf{create}\ x.m]\!]_{\mathcal{Q}} : \text{Objects} \to \text{States} \to \text{Objects}\\&[\![\textbf{create}\ x.m]\!]_{\mathcal{Q}} = \text{undefined}\\&[\![\textbf{create}\ x.m]\!]_{\mathcal{S}} : \text{Objects} \to \text{States} \to \text{States}\\&[\![\textbf{create}\ x.m]\!]_{\mathcal{S}}\ u\ \sigma = [\![x := m]\!]_{\mathcal{S}}\ u\ ([\![m]\!]_{\mathcal{S}}\ u\ (\text{alloc}_{\mathcal{Q}}\ c\ \sigma)\ (\text{alloc}_{\mathcal{S}}\ c\ \sigma))\end{aligned} \tag{3.33}$$

Note: m denotes the creation feature after argument passing. See section 3.5.11 for how arguments are passed to a feature.

3.5.6 Conditional

A conditional instruction splits the path of execution in two branches, depending on the result of evaluating a boolean expression. To shorten the notation the following function 'cond' is defined.

$$\begin{aligned}&f : \text{Objects} \to \text{States} \to \mathbb{B}\\&g, h : \text{Objects} \to \text{States} \to \mathit{States}\\&\text{cond}(f, g, h)\ u\ \sigma := \begin{cases} g\ u\ \sigma, & \text{if } f\ u\ \sigma \\ h\ u\ \sigma, & \text{otherwise}\end{cases}\end{aligned} \tag{3.34}$$

Let b be a boolean expression and let s and t be instruction sequences.

$$\begin{aligned}&[\![\textbf{if } b \textbf{ then } s \textbf{ else } t \textbf{ end}]\!]_\mathcal{Q} : \text{Objects} \to \text{States} \to \text{Objects}\\&[\![\textbf{if } b \textbf{ then } s \textbf{ else } t \textbf{ end}]\!]_\mathcal{Q} := \text{undefined}\\&[\![\textbf{if } b \textbf{ then } s \textbf{ else } t \textbf{ end}]\!]_\mathcal{S} : \text{Objects} \to \text{States} \to \text{States}\\&[\![\textbf{if } b \textbf{ then } s \textbf{ else } t \textbf{ end}]\!]_\mathcal{S} := \text{cond}([\![b]\!]_\mathcal{Q}, [\![b; s]\!]_\mathcal{S}, [\![b; t]\!]_\mathcal{S})\end{aligned} \tag{3.35}$$

3.5.7 Loop

A loop starts with an initialization part. Afterward the loop body is executed as long as the termination condition does not evaluate to true.

We define the semantics of the general loop in terms of the special case without initialization part. Let b be a boolean expression and let i and s be instruction sequences.

$$[\![\textbf{from } i \textbf{ until } b \textbf{ loop } s \textbf{ end}]\!] = [\![i; \textbf{until } b \textbf{ loop } s \textbf{ end}]\!] \tag{3.36}$$

Let $\mathit{loop} := \textbf{until } b \textbf{ loop } s \textbf{ end}$. The semantic functions of the special loop are defined as follows:

$$\begin{aligned}&[\![\mathit{loop}]\!]_\mathcal{Q} : \text{Objects} \to \text{States} \to \text{Objects}\\&[\![\mathit{loop}]\!]_\mathcal{Q} := \text{undefined}\\&[\![\mathit{loop}]\!]_\mathcal{S} : \text{Objects} \to \text{States} \to \text{States}\\&[\![\mathit{loop}]\!]_\mathcal{S} := \begin{cases} [\![b]\!]_\mathcal{S}\ u\ \sigma, & \text{if } [\![b]\!]_\mathcal{Q}\ u\ \sigma = \text{True} \\ [\![s; \mathit{loop}]\!]_\mathcal{S}\ u\ ([\![b]\!]_\mathcal{S}\ u\ \sigma), & \text{otherwise}\end{cases}\end{aligned} \tag{3.37}$$

3.5.8 Current

The 'Current' keyword just returns the object passed as target and does not modify the state.

$$\begin{aligned}&[\![\text{Current}]\!]_\mathcal{Q} : \text{Objects} \to \text{States} \to \text{Objects}\\&[\![\text{Current}]\!]_\mathcal{Q}\ u\ \sigma = u\\&[\![\text{Current}]\!]_\mathcal{S} : \text{Objects} \to \text{States} \to \text{States}\\&[\![\text{Current}]\!]_\mathcal{S}\ u\ \sigma = \sigma\end{aligned} \tag{3.38}$$

According to the syntax definition in chapter 2 a feature invocation relative to the current object is considered an instruction or a expression. However the signatures of the semantic functions of a feature are not compatible to that of an instruction or expression. Therefore we expect the compiler to always put a preceding 'Current.' to every feature invocation which is done directly on the current object.

3.5.9 The Dot-Operator

The dot-operator allows to invoke a feature g on the result object of an expression f. The feature g may be a query or a command. In the former case the result is again an expression, in the latter case the result is a command, and thus an instruction.

$$\begin{aligned}
&[\![f.g]\!]_{\mathcal{Q}} : \text{Objects} \to \text{States} \to \text{Objects} \\
&[\![f.g]\!]_{\mathcal{Q}}\ u\ \sigma := [\![g]\!]_{\mathcal{Q}}\ u\ ([\![f]\!]_{\mathcal{Q}}\ u\ \sigma)\ ([\![f]\!]_{\mathcal{S}}\ u\ \sigma) \\
&[\![f.g]\!]_{\mathcal{S}} : \text{Objects} \to \text{States} \to \text{States} \\
&[\![f.g]\!]_{\mathcal{S}}\ u\ \sigma := [\![g]\!]_{\mathcal{S}}\ u\ ([\![f]\!]_{\mathcal{Q}}\ u\ \sigma)\ ([\![f]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.39}$$

Note: g is the semantic function after argument passing. How arguments are passed to a feature is described in section 3.5.11.

3.5.10 Tuples

We need tuples to pass arguments to a feature. A tuple consists of a list of expressions which all are evaluated relative to the same object. The evaluation order is from left to right. Let $a_i : i = 1 \dots n$ be expressions.

$$\begin{aligned}
&[\![(a_1, \dots, a_n)]\!]_{\mathcal{Q}} : \text{Objects} \to \text{States} \to \text{Objects}^n \\
&[\![(a_1, \dots, a_n)]\!]_{\mathcal{S}} : \text{Objects} \to \text{States} \to \text{States}
\end{aligned} \tag{3.40}$$

Define the semantics by induction on $n \geq 1$.

$$\begin{aligned}
&[\![(a_1)]\!]_{\mathcal{Q}}\ u\ \sigma := ([\![a_1]\!]_{\mathcal{Q}}\ u\ \sigma) \\
&[\![(a_1)]\!]_{\mathcal{S}}\ u\ \sigma := [\![a_1]\!]_{\mathcal{S}}\ u\ \sigma
\end{aligned} \tag{3.41}$$

Given the semantics for n-tuples the semantics for $(n+1)$-tuples is defined as follows ($\oplus$ denotes concatenation of tuples):

$$\begin{aligned}
&[\![(a_1, \dots, a_n, a_{n+1})]\!]_{\mathcal{Q}} := \\
&\quad ([\![(a_1, \dots, a_n)]\!]_{\mathcal{Q}}\ u\ \sigma) \oplus ([\![(a_{n+1})]\!]_{\mathcal{Q}}\ u\ ([\![(a_1, \dots, a_n)]\!]_{\mathcal{S}}\ u\ \sigma) \\
&[\![(a_1, \dots, a_n, a_{n+1})]\!]_{\mathcal{S}} := [\![(a_n)]\!]_{\mathcal{S}}\ u\ ([\![(a_1, \dots, a_n)]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.42}$$

3.5.11 Argument Passing

Passing arguments to a feature is quite simple. The first argument to the semantic functions of a feature with arguments is a tuple of appropriate dimension.

The signature of the result is that of a feature without arguments. Let f be a feature that takes n arguments.

$$\begin{aligned}
&[\![f(p)]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\
&[\![f(p)]\!]_{\mathcal{Q}}\ u\ v\ \sigma := [\![f]\!]_{\mathcal{Q}}\ ([\![(p)]\!]_{\mathcal{Q}}\ u\ \sigma)\ v\ ([\![(p)]\!]_{\mathcal{S}}\ u\ \sigma)\\
&[\![f(p)]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\\
&[\![f(p)]\!]_{\mathcal{Q}}\ u\ v\ \sigma := [\![f]\!]_{\mathcal{S}}\ ([\![(p)]\!]_{\mathcal{Q}}\ u\ \sigma)\ v\ ([\![(p)]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.43}$$

3.5.12 Equality Tests

A special sort of boolean expressions are equality tests. An equality test returns 'True' if both expressions evaluate to the same object. Since expressions may have side-effects the evaluation not both expressions are evaluated relative to the same state in general. We define that first the left hand side and then the right hand side is evaluated. The inequality test is just the negation of the equality test.

Equality

Let q_1 and q_2 be expressions.

$$\begin{aligned}
&[\![q_1 = q_2]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\
&[\![q_1 = q_2]\!]_{\mathcal{Q}}\ u\ \sigma := ([\![q_1]\!]_{\mathcal{Q}}\ u\ \sigma) = ([\![q_2]\!]_{\mathcal{Q}}\ u\ ([\![q_1]\!]_{\mathcal{S}}\ u\ \sigma))\\
&[\![q_1 = q_2]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\\
&[\![q_1 = q_2]\!]_{\mathcal{S}}\ u\sigma := [\![q_2]\!]_{\mathcal{S}}\ u\ ([\![q_1]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.44}$$

Inequality

Let q_1 and q_2 be expressions.

$$\begin{aligned}
&[\![q_1 \neq q_2]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects}\\
&[\![q_1 \neq q_2]\!]_{\mathcal{Q}}\ u\ \sigma := ([\![q_1]\!]_{\mathcal{Q}}\ u\ \sigma) \neq ([\![q_2]\!]_{\mathcal{Q}}\ u\ ([\![q_1]\!]_{\mathcal{S}}\ u\ \sigma))\\
&[\![q_1 \neq q_2]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States}\\
&[\![q_1 \neq q_2]\!]_{\mathcal{S}}\ u\sigma := [\![q_2]\!]_{\mathcal{S}}\ u\ ([\![q_1]\!]_{\mathcal{S}}\ u\ \sigma)
\end{aligned} \tag{3.45}$$

3.6 Assertions

The semantic definition described so far does not take into account assertions. Whether assertions should have a runtime effect or not can be disputed. From a pure proof-oriented point of view assertions serve for specification only. A program that does not meet its specification should not be executed at all. So assertions do not have to have a runtime semantics.

A programmer on the other hand may want the assertions to be checked at runtime in the testing phase. An assertion violation can give important hints to detect errors. Without checked assertions an error may manifests quite anywhere in the execution, but usually not at the origin. Thus checked assertions allow to detect violations early and trace back to the origin efficiently.

As pointed out above a correct program does not need to test assertions at runtime anymore. Hence it should be possible to compile a program so that assertion checking is switched off. In this section we complete the semantic definition for the case when assertion checking is active.

In our language we have three kinds of assertions: Preconditions, postconditions and checks. The signature of the semantic evaluation for assertions is as follows. Let a be an assertion.

$$\begin{aligned} &[\![a]\!]_{\mathcal{Q}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{Objects} \\ &[\![a]\!]_{\mathcal{S}} : \text{Objects} \rightarrow \text{States} \rightarrow \text{States} \end{aligned} \tag{3.46}$$

The query semantics of an assertions returns a boolean value which is 'true' if the assertion is satisfied and 'false' otherwise. In the case an assertion fails the program should stop and indicate a failure. This is done by a special failure state Fail $\in$ State.

$$[\![a]\!]_{\mathcal{Q}}\ u\ \sigma = \text{false} \ \Rightarrow\ [\![a]\!]_{\mathcal{S}}\ u\ \sigma = \text{Fail} \tag{3.47}$$

3.6.1 Precondition

A precondition is a simple boolean expression. In a feature invocation first the state is initialized as described in section 3.4.3. Then the precondition is evaluated. Before execution of the body starts the postcondition has to be inspected for expressions which refer to the prestate, marked with the 'old' keyword. For each such expression a special local variable is allocated and the value of the expression is evaluated and assigned to this variable. The order of evaluation is the order of occurrence in the postcondition. Expressions which appear more than once are evaluated repeatedly and also stored in different variables. If side-effect-freeness of all expressions in assertions can be guaranteed, the order of evaluation does not matter. As already stated earlier we will only consider programs with side-effect-free assertions.

3.6.2 Postcondition

A postcondition is a boolean expression where expressions with the modifier 'old' may occur. This indicates that the value of the expression before execution of the feature has to be replaced there. As described above the old values are stored in designated local variables. The postcondition is evaluated right after the execution of the body, before the caller's context is restored.

Additionally to the postcondition also the invariants have to be checked to guarantee that the object is in a consistent state.

3.6.3 Checks

A check consists of a boolean expression. Syntactically a check is an instruction. A check can e.g. be used to emulate a loop invariant, which is not supported directly by a designated construct.

Chapter 4

Applications

The semantic definition presented in the last chapter is kept rather short in order to focus on the essential. This approach is admissible, because the concepts supported are not new inventions but rather classical constructs. Their general purpose often is explained in a few words. The aim of a formal semantic definition is to translate the *natural* meaning into a formal system. Much aspects of the intuitive understanding of program turn out to leave freedom for interpretation. Thus an important contribution of a formal definition is to appoint a unique meaning to a program. However by now we gave less justification that our definition represents the natural meaning well.

A formal semantics allows to reason whether a program has a certain property or not. The semantics is called *sound* if for every program p and every property P at most one of the following statements can be proved using the formal semantics:

1. p has the property P
2. p does not have the property P.

Obviously a semantic definition which allows to derive contradicting statements is of no use. Anything can be proved or disproved with such a semantics. Hence soundness is essential for a semantic definition.

In this chapter we want to validate the semantic definition by an example application. Though this does certainly not prove soundness it may increase the confidence in the semantic definition. Furthermore it may uncover weaknesses of the language's expressiveness with respect to program verification.

As an example application we choose an implementation of a stack using a linked list. In fact linked lists and stacks are favored data structures in the literature. See e.g. [6] or [9]. Their implementations are simple enough to keep the example small, yet not trivial as their structures evolves dynamically during execution.

4.1 Class LINKABLE

The abstraction of a linked list is a finite sequence of elements of some type G. In order to implement such a list one need additional objects which store the information about the linking. Thus a linkable object consists of two references. One to the item it stores, and another one to the next linkable object in the sequence. If there is no subsequent linkable object the second reference is void.

In the following implementation the features 'item', 'set_item', 'next' and 'set_next' have the obvious meaning.

4.1.1 Implementation

```
class LINKABLE
creation make
feature {ANY}
  next : LINKABLE

  set_next(n : LINKABLE) is
  do
    next := n
  ensure
    next = old n
  end

  item : G

  set_item(i : G) is
  do
    item = i
  ensure
    item = old i
  end

  make is
  do
    next := Void;
    item := Void
  ensure
    (next = Void).and(item = Void)
  end
end
```

4.1.2 Correctness

Before we start to prove correctness we have to be aware what correctness means.

> A class is called *correct* if none of its feature cause a runtime error due to a violation of an assertion or a feature invocation on the object 'Void'.

In order to guarantee this we have the following proof obligations:

1. Every feature fulfills its postcondition and the class invariant, provided the precondition is satisfied.
2. For every feature invocation the target object is not 'Void'.
3. For every feature invocation the precondition of the called feature is fulfilled.

We restricted ourselves to programs for which the evaluation of assertions do not have side-effects. Therefore we also have to ensure that our examples meet this restriction.

'item' and 'next'

The features 'item' and 'next' are attributes. Hence they do not have contracts, and we have nothing to show. We note that invocations of attributes are side-effect free, according to (3.13). Hence they can be used in assertions.

'set_next'

First we convince ourselves that all assertions are side-effect free. As mentioned above 'next' is an attribute. 'n' is a formal argument, which is represented by a local variable according to section 3.4.3. Thus all expressions in the assertions are side-effect free.

We have to show that for an arbitrary state σ and every object $u \in$ LINKABLE which is allocated in state σ the following hold:

$$[\![\text{next} = \textbf{old}\ \text{n}]\!]_{\mathcal{Q}}\ \text{u}\ ([\![\text{next} := \text{n}]\!]_{\mathcal{S}}\ \text{u}\ \sigma) = \text{true}. \tag{4.1}$$

In order to do this we need some facts. In section (3.6.2) it is stated that an expression preceded by 'old' is evaluated prior execution of the body. The result is stored in a special local variable, not accessible by the programmer. Hence the value of this variable is preserved during computation, and we can conclude:

$$[\![\textbf{old}\ n]\!]_{\mathcal{Q}}\ u\ ([\![\text{next} := \text{n}]\!]_{\mathcal{S}}\ u\ \sigma) = [\![\text{n}]\!]_{\mathcal{Q}}\ \text{u}\ \sigma. \tag{4.2}$$

We also want to compute the value of 'next' after the body is executed.

$$\begin{aligned} & [\![\text{next}]\!]_{\mathcal{Q}}\ \text{u}\ ([\![\text{next} := \text{n}]\!]_{\mathcal{S}}\ \text{u}\ \sigma) \\ & = \text{attributes}\ ([\![\text{next} := \text{n}]\!]_{\mathcal{S}}\ u\ \sigma)\ u\ \text{next} \quad \text{by (3.13)} \\ & = [\![\text{n}]\!]_{\mathcal{Q}}\ \text{u}\ \sigma \quad \text{by (3.29)} \end{aligned} \tag{4.3}$$

By combining the last two equations we get:

$$[\![\text{next}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{next} := \text{n}]\!]_{\mathcal{S}} \text{ u } \sigma) = [\![\textbf{old} \text{ n}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{next} := \text{n}]\!]_{\mathcal{S}} \text{ u } \sigma). \tag{4.4}$$

For abbreviation we define:

$$\sigma_1 := [\![\text{next} := \text{n}]\!]_{\mathcal{S}}\ u\ \sigma. \tag{4.5}$$

With the definition of an equality test (3.44) and side-effect-freeness we can deduce:

$$\begin{array}{ll} [\![\text{next} = \textbf{old} \text{ n}]\!]_{\mathcal{Q}} \text{ u } \sigma_1 & \text{by } (3.44) \\ = ([\![\text{next}]\!]_{\mathcal{Q}} \text{ u } \sigma_1 = [\![\textbf{old} \text{ n}]\!]_{\mathcal{Q}} \text{ u } \sigma_1) & \text{by } (4.4) \\ = \text{true}. & \end{array} \tag{4.6}$$

Hence the postcondition is satisfied. The proof for the feature 'set_item' is analogous.

Correctness of 'make'

Again we make sure that the expressions in the postcondition are side-effect free. This allows us to proof the two equations in the conjunction separately, since we also know that the logic operations have no side-effects.

We make the same assumption as above. Let σ be a state and $u \in LINKABLE$ an allocated object in σ. First let us prove

$$[\![\text{next} = \text{Void}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{next} := \text{Void}; \text{item} := \text{Void}]\!]_{\mathcal{S}} \text{ u } \sigma) = \text{true}. \tag{4.7}$$

Using (3.25) we define:

$$\begin{array}{l} \sigma_1 := [\![\text{next} := \text{Void}]\!]_{\mathcal{S}} \text{ u } \sigma \\ \sigma_2 := [\![\text{item} := \text{Void}]\!]_{\mathcal{S}} \text{ u } \sigma_1. \end{array} \tag{4.8}$$

By (3.29) and the fact that the evaluation of 'Void' is side-effect-free we can conclude:

$$\begin{array}{l} [\![\text{next}]\!]_{\mathcal{Q}} \text{ u } \sigma_1 = \text{Void} \quad \text{and} \\ [\![\text{next}]\!]_{\mathcal{Q}} \text{ u } \sigma_2 = \text{Void} \end{array} \tag{4.9}$$

Hence by (3.44) we have:

$$[\![\text{next} = \text{Void}]\!]_{\mathcal{Q}} \text{ u } \sigma_2 = \text{true} \tag{4.10}$$

which we wanted to prove.

We still have to prove

$$[\![\text{item} = \text{Void}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{next} := \text{Void}; \text{item} := \text{Void}]\!]_{\mathcal{S}} \text{ u } \sigma) = \text{true}. \tag{4.11}$$

Again we use (3.29):

$$[\![\text{item}]\!]_{\mathcal{Q}} \text{ u } \sigma_2 = \text{Void} \tag{4.12}$$

Using (3.44) we get:

$$[\![\text{item} = \text{Void}]\!]_{\mathcal{Q}} \text{ u } \sigma_2 \tag{4.13}$$

and we are done. Thus all features of class LINKABLE satisfy their contracts.

4.2 Class STACK

A stack is a so called *first in first out* data structure. There are operations which allow to put elements on the stack, to get the last element put and to remove the last element put. In general the order in which elements are put on the stack is the reverse order in which they are removed. The top of the stack is the element most recently put.

empty: Returns 'True' if the stack is empty and 'False' otherwise.

put: Inserts an item of type G on top of the stack.

item: Returns the element on top of the stack.

remove: Removes the element on top of the stack

4.2.1 Implementation

```
class STACK
creation make
feature {ANY}
  empty : BOOLEAN is
  do
    Result := first = Void
  end

  item : G is
  require
    empty.not
  do
    Result := first.item
  end

  put(x : G) is
  local new : LINKABLE
  do
    create new.make;
    new.set_item(x);
    new.set_next(first);
    first := new
  ensure
    top = old x
  end
```

```
remove is
  require
    empty.not
  do
    first = first.next
  end

  make is
  do
    first := Void
  ensure
    empty
  end

feature {NONE}
  first : LINKABLE
end
```

4.2.2 Correctness

'empty'

The feature 'empty' does not have a contract, thus it fulfills it trivially. But the feature is used in contracts of other features. Therefore we have to prove that 'empty' is side-effect free. Let σ be some state and $u \in$ STACK an object which is allocated in state σ.

In order to prove that some feature f is side-effect free, one has to show that for $\sigma' := [\![f]\!]_{\mathcal{S}}\ u\ \sigma$ the following hold:

$$\text{attributes } \sigma' = \text{attributes } \sigma. \tag{4.14}$$

To be precise one also should show that no new objects were allocated. However in 'empty' no creation instruction occurs.

Let us denote the state after execution of the body by σ':

$$\sigma' := [\![\text{Result} := \text{first} = \text{Void}]\!]_{\mathcal{S}}\ \text{u}\ \sigma. \tag{4.15}$$

Then by (3.27) we have to show:

$$\text{attributes } \sigma' = \text{attributes } ([\![\text{first} = \text{Void}]\!]_{\mathcal{S}}\ \text{u}\ \sigma). \tag{4.16}$$

It suffices to show that

$$[\![\text{first} = \text{Void}]\!]_{\mathcal{S}}\ \text{u}\ \sigma = \sigma \tag{4.17}$$

We know that the expressions on both sides of the equality sign are side-effect free (Note: 'first' is an attribute). Thus by (3.44) we can write:

$$\begin{aligned}&[\![\text{first} = \text{Void}]\!]_{\mathcal{S}}\ \text{u}\ \sigma\\ &= [\![\text{Void}]\!]_{\mathcal{S}}\ \text{u}\ ([\![\text{first}]\!]_{\mathcal{S}}\ \text{u}\ \sigma)\\ &= [\![\text{Void}]\!]_{\mathcal{S}}\ \text{u}\ \sigma\\ &= \sigma.\end{aligned} \tag{4.18}$$

Hence 'empty' is side-effect free with respect to the object structure.

'item'

The feature 'item' does not have a postcondition. What we have to prove is that the feature call 'first.item' does not fail. Again σ is an arbitrary state and $u \in$ STACK is an object which is allocated in σ. We have to show:

$$[\![\text{empty.not}]\!]_{\mathcal{Q}}\ \text{u}\ \sigma = \text{true}\ \Rightarrow\ [\![\text{first}]\!]_{\mathcal{Q}}\ \text{u}\ \sigma \neq \text{Void}. \tag{4.19}$$

Since 'empty' is a feature of the same class we are allowed to look at the implementation of 'empty' without violating information hiding. We will execute 'empty' and try to deduce the property (4.19). Side-effect-freeness of 'first' and 'Void' will be used without further reference.

$$\begin{array}{ll} [\![\text{empty}]\!]_{\mathcal{Q}}\ \text{u}\ \sigma & \\ = \text{locals}\ ([\![\text{Result} := \text{first} = \text{Void}]\!]_{\mathcal{S}}\ \text{u}\ (\text{init}\ \sigma))\ \text{Result} & \text{by (3.17)} \\ = [\![\text{first} = \text{Void}]\!]_{\mathcal{Q}}\ \text{u}\ (\text{init}\ \sigma) & \text{by (3.27)} \\ = ([\![\text{first}]\!]_{\mathcal{Q}}\ \text{u}\ (\text{init}\ \sigma) = [\![\text{Void}]\!]_{\mathcal{Q}}\ \text{u}\ (\text{init}\ \sigma)) & \text{by (3.44)} \\ = (\text{attributes}\ (\text{init}\ \sigma)\ u\ \text{first} = \text{Void}) & \text{by (3.13)} \\ = (\text{attributes}\ \sigma\ u\ \text{first} = \text{Void}) & \text{by (3.14)} \\ = ([\![\text{first}]\!]_{\mathcal{Q}}\ \text{u}\ \sigma = \text{Void}) & \text{by (3.13)} \end{array} \tag{4.20}$$

Hence 'empty' returns true if and only if 'first' is a void reference. This statement is even stronger than the one we wanted to prove.

With this result we get the correctness proofs for 'remove' and 'make' for free. Feature 'remove' can be treated the same as feature 'item'. And in feature 'make' we see at once that 'first' is assigned to a void reference, implying that the postcondition will be satisfied.

'push'

To do the tracing for 'push' by hand becomes already very complicated. We will try it all the same as it exhibits weaknesses of our assertions for the issue of specification.

We have to show for any state σ and object $u \in$ STACK which is allocated in σ, that the postcondition is satisfied. First let us abbreviate the intermediate states of execution using (3.25):

$$\begin{array}{l} \sigma_1 := [\![\textbf{create}\ \text{new.make}]\!]_{\mathcal{S}}\ \text{u}\ \sigma \\ \sigma_2 := [\![\text{new.set\_item(x)}]\!]_{\mathcal{S}}\ \text{u}\ \sigma_1 \\ \sigma_3 := [\![\text{new.set\_next(first)}]\!]_{\mathcal{S}}\ \text{u}\ \sigma_2 \\ \sigma_4 := [\![\text{first} := \text{new}]\!]_{\mathcal{S}}\ \text{u}\ \sigma_3. \end{array} \tag{4.21}$$

Then the proof obligation can be stated as follows:

$$[\![\text{top} = \textbf{old}\ \text{x}]\!]_{\mathcal{Q}}\ \text{u}\ \sigma_4 = \text{True} \tag{4.22}$$

The old value of 'x' can be represented as follows:

$$[\![\textbf{old } x]\!]_{\mathcal{Q}}\ u\ \sigma_4 = [\![x]\!]_{\mathcal{Q}}\ u\ \sigma. \tag{4.23}$$

Now we try to evaluate 'top' in the postcondition:

$$\begin{aligned}
&[\![top]\!]_{\mathcal{Q}}\ u\ \sigma_4\\
&\quad \{\text{by (3.17)}\}\\
&= \text{locals } ([\![\text{Result := first.item}]\!]_{\mathcal{S}} \text{ u (init } \sigma_4)) \text{ Result}\\
&\quad \{\text{by (3.27)}\}\\
&= [\![\text{first.item}]\!]_{\mathcal{Q}} \text{ u (init } \sigma_4)\\
&\quad \{\text{by (3.39) and side-effect-freeness of 'first'}\}\\
&= [\![\text{item}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{first}]\!]_{\mathcal{Q}} \text{ u (init } \sigma_4)) \text{ (init } \sigma_4)\\
&\quad \{\text{'init' does not change the 'attributes' component}\\
&\quad\ \text{and 'item' and 'first' are attributes}\}\\
&= [\![\text{item}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{first}]\!]_{\mathcal{Q}} \text{ u } \sigma_4)\ \sigma_4\\
&\quad \{\text{postcondition of 'set\_next'}\}\\
&= [\![\text{item}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{new}]\!]_{\mathcal{Q}} \text{ u } \sigma_3)\ \sigma_4\\
&\quad \{\text{the assignment does not change 'item'}\}\\
&= [\![\text{item}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{new}]\!]_{\mathcal{Q}} \text{ u } \sigma_3)\ \sigma_3\\
&\quad \{\text{'new' is a local variable and hence not changed by 'set\_next'}\}\\
&= [\![\text{item}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{new}]\!]_{\mathcal{Q}} \text{ u } \sigma_2)\ \sigma_3\\
&\quad \{\text{assume that 'set\_next' does not change 'item'}\}\\
&= [\![\text{item}]\!]_{\mathcal{Q}} \text{ u } ([\![\text{new}]\!]_{\mathcal{Q}} \text{ u } \sigma_2)\ \sigma_2\\
&\quad \{\text{postcondition of 'set\_item'}\}\\
&= [\![x]\!]_{\mathcal{Q}}\ u\ \sigma_1\\
&\quad \{\text{'x' is not changed by the creation of 'new'}\}\\
&= [\![x]\!]_{\mathcal{Q}}\ u\ \sigma
\end{aligned} \tag{4.24}$$

As we can see both sides of the equation evaluate to the same object, therefore the precondition is fulfilled. However we had to make the assumption that the feature invocation of 'set_next' does not change the value of 'item' in the same object. The postcondition of 'set_next' should be extended to ensure this. Thus proofs may be helpful to detect incomplete specifications. On the other hand this example demonstrates how important it is to state assertions carefully.

4.3 Conclusions

On one hand the proof examples show that the semantics can be used to derive some correctness properties from a program. However this does not imply that the semantics is sound. On the other hand also weaknesses of the semantics, and also the language become apparent. They are discussed in the remainder of this section.

4.3.1 Proof Automation

We used the formal definitions extensively in the proofs. However major parts of the proofs are done by reasoning in natural language. In order to support automated proof verification this proofs have to be stated in a formal calculus. At the current stage it is not clear whether a translation of the semantic definition in a purely formal system is possible. A subsequent step may be to choose a proving environment and try to translate the semantic definition to this environment.

4.3.2 Side-effect in Assertions

We always had to ensure that the expressions used in the assertions are side-effect free. In the example of the 'empty'-feature this was a tedious work. We only accomplished it because we knew the one and only implementation. In a more general setting a query used in an assertion could have several implementations. Either we have to give up information hiding to prove side-effect-freeness or state this property in the contract of a feature.

4.3.3 Expressiveness of Assertions

The contracts used to specify class 'STACK' are not sufficient to describe the behavior of a stack completely. E.g. the feature 'remove' does not even provide a postcondition. What can a client expect after an invocation of 'remove'? According to this specification it is not forbidden to remove all elements at once. To indicate what a feature does not change is completely out of scope in an open world. In a closed world it might force the programmer to extend the contracts from time to time.

Chapter 5

Summary

In the chapters 2 and 3 we defined an object-oriented example language. Most of its features of this language are borrowed from *Eiffel*, as it is one of the few languages which allows to state parts of the specifications in the program code.

We already pointed out how important good specifications are. On one hand they allow to validate implementations according to clearly stated criteria. On the other hand they are an indispensable means for modular verification. When in a program a feature of some foreign class is invoked, information hiding prohibits to consult the features implementation. The specification is the only source of information about the effect of a feature invocation. So we have to discuss whether the contract approach suffices for specification.

5.1 Analysis

The aim of chapter 4 was to exploit the limitations of the expressiveness of assertions. We accomplished to prove that the given stack implementation does fulfill its contracts, though these contracts are far from specifying the stack fully. Moreover some features could be easily proved to be correct simply because they do not have any postcondition at all. This may be good news to the person which has to prove correctness of such a feature. But such correctness proofs do not turn out to be very valuable for reuse. A poor specification make it very difficult to prove correctness of a client which has to rely on such a specification.

Another serious problem are potential side-effects of assertion. Although we restricted ourselves only to allow side-effect-free assertions this cannot be the final solution. As we have seen to check side-effect-freeness may not be trivial.

Some of the proofs became quite complex, although the complexity of the implementations was rather small. So it may be doubted that the provided semantics is suited to verify more complex programs.

The main points to be discussed here are:

1. How can assertions be made more expressive, so that data types can be specified adequately?

2. How can one enforce assertions not to have side-effects?

3. How can the semantic definition be improved to support automated proof verification?

5.2 Models

In the application of chapter 4 the limitations to specify a data structure completely became apparent. In the assertions we only allow expressions to occur. Thus everything which can be evaluated in an assertions must be implemented as a feature of some class. In order to specify a stack we would need a feature which returns a representation of the entire stack. This is not possible with the interface of the stack which only allows to access the top element. It also would contradict the stack abstraction. For a client it should not be allowed to access arbitrary elements on the stack.

What we need is some kind of abstraction variable, as proposed by Hoare [1], which represents the state of a stack in an abstract way. In addition the stack has to provide a feature which returns the stack's abstract state.

Meyer [9], and subsequently Schoeller [14], propose to use *models* to represent data types in an abstract way. One of the main advantages is that models can be implemented as ordinary classes and does not have to introduce new language constructs.

Using *Eiffel's* selective export mechanism the feature which maps the concrete representation to the corresponding abstract model can be made accessible exclusively to the model.

5.3 Side-Effects in Assertions

The model approach sketched above does not solve the problem with side-effects of assertions. Though the model objects are supposed to be immutable, i.e. their state cannot be changed by feature invocations. However it is not guaranteed that such an invocation is side-effect free. Furthermore the query to obtain the model object from the implementation should not have side-effects, except for the creation of the returned model object. Again this property cannot be enforced and thus quite much responsibility is burdened to the programmer. The point of side-effects is also objected against executable contracts by Poetzsch-Heffter [13]. Schoeller [14] argues that the disadvantages can be avoided with appropriate constraints on expressions in contracts. Moreover the possibility to use language constructs instead of first-order predicate logic to state specification eliminates the conceptual gap between specification language and implementation language.

5.4 Conclusion and Outlook

The motivation for this project was to explore the possibilities of program verification in the object-oriented paradigm. As explained in the introduction due to its high degree of modularity and abstraction object-orientation seems predestined for efficient program verification.

The first step of such an investigation consists in defining a formal semantics of an object-oriented language. It turned out that the ambitious goal of modular verification cannot be achieved just by providing a clever semantics. In contrast it became clear that the frame problem is one of the main obstacles for modular proofs in an open environment.

In a future work the semantics defined in this project should be translated to a formal proof environment. Furthermore it should be considered whether it may serve as a basis to define an axiomatic semantics, which may be better suited for program verification. All the more because contracts also are stated in an axiomatic style.

And last but not least one should consider if the results of Müller, Poetzsch-Heffter and Leino could be adapted for the design by contract method.

Bibliography

[1] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[2] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.*, 24(5):491–553, 2002.

[3] K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.

[4] John McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP-62*, 1962.

[5] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[6] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2 edition, 1997.

[7] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[8] Bertrand Meyer. *Eiffel: the language.* Prentice Hall, New York, NY, 1992.

[9] Bertrand Meyer. Towards practical proofs of class correctness. In Didier Bert et al., editor, *ZB 2003: Formal Specification and Development in Z and B*, LNCS 2651, pages pp. 359–387. Springer-Verlag Berlin, 2003.

[10] Harlan D. Mills. How to write correct programs and know it. *SIGPLAN Not.*, 10(6):363–370, 1975.

[11] P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001. Also as LNCS 2262, Springer-Verlag, 2002.

[12] Peter Müller and Arnd Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. *Foundations of component-based systems*, pages 137–159, 2000.

[13] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Univeristt Mnchen, January 1997.

[14] Bernd Schoeller. Strengthening eiffel contracts using models. Technical report, Swiss Federal Institute of Technology, Chair of Software Engineering, Zurich, Switzerland, 2003.

[15] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 26(1):70–74, 1983.