Fakultät für Informatik
Technische Universität München

# Specification and Verification of Object–Oriented Programs

Habilitationsschrift

vorgelegt von

Arnd Poetzsch-Heffter

Januar 1997

## Abstract

Interface specifications should express program properties in a formal, declarative, and implementation–independent way. To achieve implementation–independency, interface specifications have to support data abstraction. Program verification should enable to prove implementations correct w.r.t. such interface specifications. The presented work bridges the gap between existing specification and verification techniques for object–oriented programs. The integration is done within a formal framework for interface specifications and programming language semantics.

Interface specification techniques are enhanced to support the specification of data structure sharing and destructive updating of shared variables. These extensions are necessary for the specification of real life software libraries. Moreover this generalization is needed for intermediate steps in correctness proofs. For verification, Hoare logic is extended to capture recursive classes and subtyping. Based on this extended logic, techniques are presented for proving typing properties, class and method invariants.

The new techniques are developed as a foundation for programming environments supporting specification and verification of programs. A major application will be the specification and verification of reusable class libraries. That is one of the reasons why the work focuses on object-oriented languages, even though the techniques are applicable to other imperative languages as well.

**Acknowledgments**

Jürgen Eickel stimulated my interest in and ambitions for the construction of software tools based on solid mathematical foundations. I thank him for his long support at the Technical University of Munich, for his confidence, and for several fruitful discussions. Manfred Broy read through a draft version of this thesis and gave many helpful suggestions for improving it. My thanks go to him also for his work in the habilitation committee. I am also indebted to Hans Langmaack for participating in that committee.

Special thanks go to all my colleagues for an encouraging working atmosphere and a lot of fruitful discussions. Bernhard Bauer, Franz Haßmann, Mariane Hargitay, Riitta Höllerer, Barbara König, Walter Lahner, Frank Lonczewski, Christoph Pleier, Boris Reichel, Siegfried and Werner Schreiber, Hans Wittner, thank you all. I like to express my special appreciation to Peter Müller for his valuable comments on several drafts of this thesis.

This work would not have been possible without the great financial contributions of the bavarian people for their universities.

# Contents

**1 Introduction 1**

1.1 Specification and Verification of Programs . . . 1
1.2 Approach, Outline, Contributions . . . 6
1.3 Formal Background . . . 9

**2 Specifying Interfaces 11**

2.1 Techniques for Interface Descriptions . . . 11
2.2 Declarative Interface Specifications . . . 17
2.2.1 Introduction to Declarative Interface Specifications . . . 17
2.2.2 A Systematic View to Interface Specification . . . 27
2.3 Interface Specifications and Subtyping . . . 29
2.3.1 Syntactic Subtyping . . . 30
2.3.2 Semantic Subtyping . . . 32

**3 Foundation of Specification and Verification 36**

3.1 An Imperative Class–Based Kernel Language . . . 36
3.1.1 The Syntax of the Kernel Language . . . 36
3.1.2 Data and State Model of the Kernel Language . . . 41
3.1.3 Axiomatic Semantics of the Kernel Language . . . 48
3.2 Properties of Object Environments . . . 52
3.3 Programming Logic . . . 56
3.4 Metarules: Typing and Invariants . . . 63

**4 Specifying and Verifying OO–Programs 73**

4.1 Object–Oriented Programs . . . 73
4.1.1 A Kernel Language with Subtyping and Encapsulation . . . 73
4.1.2 Data and State Model for AICKL* . . . 84
4.1.3 Axiomatic Semantics for AICKL* . . . 86
4.2 Specifying Object–Oriented Programs . . . 93
4.2.1 Specifying Program Properties . . . 94
4.2.2 The Meaning of Interface Specifications . . . 97
4.2.3 Discussing Interface Specification Constructs . . . 100
4.2.4 Revisiting Interface Specifications . . . 109
4.3 Verifying Object-Oriented Programs . . . 117
4.3.1 Using Interface Specifications for Verification . . . 117

4.3.2 Verifying Invariant Properties . . . . . . . . . . . . . . . . . . . . . 122
4.3.3 Exploiting Sharing Properties . . . . . . . . . . . . . . . . . . . . . 128

**5 Conclusions 130**

**A Models, Summaries, Proofs, Examples 134**
A.1 Model for the Environment Axioms . . . . . . . . . . . . . . . . . . . . . . 134
A.2 Summary of Programming Logic . . . . . . . . . . . . . . . . . . . . . . . . 137
A.3 Proof of Lemma 3.4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
A.4 Proof of Lemma 3.13 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
A.5 Semantical Aspects of Subclassing . . . . . . . . . . . . . . . . . . . . . . 143
A.6 Implementation of Class PLIST . . . . . . . . . . . . . . . . . . . . . . . 143
A.7 Verification of a Sorting Method . . . . . . . . . . . . . . . . . . . . . . 146

**References 148**

**Index 152**

# Chapter 1

# Introduction

Making computing systems reliable is one of the central challenges for computer science. Reliability is important from an economical point of view and crucial in safety critical systems like aircraft or traffic control systems and high-tech medical tools. Realistic computing systems are built from quite a number of hardware and software components. The goal is to derive the reliability of computing systems from the reliability of their components. Such a compositional approach needs concise interface specifications for components. Interface specifications have to support abstraction and information hiding. Both properties are essential to keep the specification size for composed systems tractable and to make components implementation–independent. Implementation–independency is needed to enable the exchange of components with equivalent interface behaviour.

This thesis focuses on techniques to improve the reliability of software modules. A module in the sense of this thesis is a collection of types and procedures declared in a sequential object–oriented programming language. A formal specification and verification framework for such modules is presented. The framework is developed as a foundation for program environments that integrate design, specification, development, and verification of object–oriented program modules. It supports declarative, implementation-independent interface specifications and can handle sharing of data representations and destructive updates of data components.

The rest of this introduction is structured as follows. Section 1.1 sketches the related scientific work, i.e. the roots of the presented approach and the environment in which it has grown up. Section 1.2 provides an outline of the thesis and gives an overview over the contributions. Finally, section 1.3 summarizes the formal notations used in the following chapters.

## 1.1 Specification and Verification of Programs

Specification and verification of programs has been a research topic for at least thirty years. Several thousand research reports, articles and books have been published in this area[1]. This section surveys the historical roots of the presented thesis and

[1]The contribution about Hoare logic ([Cou90]) in the handbook of theoretical computer science lists 410 references only about this subtopic.

discusses scientific work that is relevant either because we built upon it or because it is needed to complement the presented approach.

**The Vision** The vision underlying this work is that efficient programs can be constructed by interfacing well–defined software components in such a way that the correctness and properties of the constructed programs can be derived from the specifications of the components. Such software components may implement basic data structures and algorithms, they may be toolboxes for user interface construction or networking, or they may encapsulate whole systems like e.g. data bases, to name only some examples. To turn this vision into practical techniques and tools, we need:

1. Powerful programming languages that (a) enable to write components with a formally defined semantics, that (b) provide sufficient parameterization mechanisms so that flexible instantiation patterns can be supported, and that (c) support the well–defined composition of components.

2. Interface specification languages that enable the formal specification of component behaviour in an abstract, implementation–independent way.

3. Programming logics to verify that components satisfy their interface specifications.

4. Universal specification and verification frameworks to relate interface specifications of programs to program–independent specifications so that the program construction process can be embedded into the surrounding software engineering process.

For practical applications it is not sufficient to have satisfactory solutions in these four areas. The solutions have to be integrated and they have to be supported by powerful tools. Tools supporting these four aspects in an integrated way are called *logic–based programming environments* within this thesis.

**Historical Roots** From a computer science point of view[2] the foundation of the area of program specification and verification was laid in the late sixties and the beginning seventies. Floyd and Hoare wrote their seminal papers ([Flo67] and [Hoa69]) on programming logic. Hoare and Wirth used the logical techniques to give an axiomatic semantics for (a large subset of) the programming language Pascal ([HW73]). Hoare published his influential paper [Hoa72] about data abstraction. Algebraic techniques for the specification of abstract data types were developed (cf. e.g. [LZ75, Gut75]).

In parallel to this fundamental work the first program verification tools were designed and implemented. The most advanced system at that time was the Stanford Pascal Verifier ([SVG79]). It supported almost full Pascal, in particular dynamically allocated data structures with pointers (cf. [Suz80] for the programming logic and [Pol81] for a large application). The Stanford Pascal Verifier demonstrated that computer–based tools can make programming logics applicable even to large programs. In that sense, it was successful. However, it remained a research prototype.

[2]The underlying logical and algebraic foundations are of course much older.

What was missing at that time to build a more practical tool and what progress has been made since then? We see progress in almost all aspects relevant to approach the vision sketched above and in particular relevant for the construction of logic–based programming environments:

- Modern programming languages provide a better support for modularization and encapsulation. Most object–oriented languages replace direct pointer manipulation by a more abstract concept.
- Program verification can only be successful if the interface properties of programs can adequately be specified. Interface specification languages have been developed for that purpose bridging the gap between program–independent specifications and program texts.
- Powerful formal specification techniques and specification languages are available today to specify abstract data types and software requirements in a declarative way. Interface specifications of programs have to refer to such program–independent specifications in order to achieve implementation–independency.
- Verification techniques and tools have been improved considerably and enable to automatically eliminate all simple proof steps. (Of course, this progress would not have been achieved without the tremendously increased computing power.)

The following paragraphs survey a selection of the literature constituting the progress in these areas. For the construction of logic–based programming environments the advanced techniques to generate language–specific tools from specifications are as well important (cf. e.g. [RT89, Kli93]).

**Programming Concepts** Since the days of Pascal, three programming concepts have played a prominent role in the development and realization of modern imperative programming languages:

1. Modularization, encapsulation: Modularization enables to compose programs from several parts with precise syntactic interfaces. Syntactic interface properties are automatically checked, in particular typing properties. Modularization goes along with (a) separation between interfaces and implementations and (b) encapsulation of implementation parts on the level of modules. One of the first programming languages supporting modularization was Modula–2 (cf. [Wir85]). Most object–oriented programming languages support in addition encapsulation on the level of types (cf. [JG96]). Encapsulation is important for specification and verification of software components because certain interface properties can only be guaranteed if the access to the implementation can be restricted.
2. Extendibility, adaptability: Beside the mere composition of fixed modules, programming languages should enable to extend and adapt program components. Modern programming languages provide two different techniques for this purpose. *Genericity* supports the development of parameterized components that

can be adapted to special needs by instantiating the parameters (language constructs for genericity can be e.g. found in Ada, cf. [Ada83], and C++, cf. [ES90]). *Subtyping* and *subclassing* enables the specialization of interfaces and implementations by overwriting or modifying procedures and by adding new variables, attributes, and procedures (cf. subsection 4.1.1.3). A comparison of these two techniques can be found in [Mey88], chapter 19. A combination of both is described in [OW97]. Adaptability of software components is in particular important to capture complex interfaces (like e.g. those of toolkits for user interface construction) by the means of the programming language.

3. Concurrency, distributivity: Many modern programming languages support constructs for writing concurrent and distributed software. E.g. Ada provides language constructs for programming concurrent tasks similar to CSP and Java supports threads, a corresponding synchronization concept, and predeclared classes to write distributed programs. Concurrency and distributivity are beyond the scope of this thesis. However, many techniques developed in the thesis are applicable to and important for concurrent and distributed programs as well.

The techniques presented in this thesis are developed for a fairly large and practically important language class including Ada 95, C++, Java, Eiffel, Sather, Oberon, Modula-3, Simula, and BETA. Although different in many aspects, all of them claim to be object–oriented.

**Interfacing Specifications and Programs** From a theoretical point of view, a program is a special kind of specification, namely a specification that can be executed on computers (cf. e.g. [Mor94] for a discussion). The software development practice nowadays is in a different situation. There are two, only loosely related worlds: on one side informal and formal specifications, on the other side programs written in complex programming languages (in most cases without a formal semantics). We distinguish two different approaches to bridge this gap. The first approach introduces special interface specification languages for this purpose. The second approach aims to integrate specification and programming in one wide–spectrum language.

**Interface Specification Languages** Interface specifications should relate the operational, state–based world of programs and the declarative, state–less world of universal specifications. Interfaces are usually specified by invariants for types and pre– and postconditions for procedures. The simplest approach is to specify properties of programs by boolean expressions of the underlying programming language. A typical example of this technique are the assertions in Eiffel (cf. [Mey92], chapter 9). Based on this technique, interface specifications have to be expressed in terms of data types and procedures of the programming language. This leads to implementation–dependent specifications and does not support abstraction at the interface. Anna, an interface specification language for Ada (cf. [Luc90]), used a combination of boolean expressions and declarative concepts to overcome these disadvantages.

The Larch approach to specification (cf. [GH93]) goes a step further. It separates programs, interface specifications, and specifications written in the declarative and programming–language–independent specification language LSL. Larch interface specifications may refer to the program–state, but must not use procedures of the underlying programming language. To express program properties, they are based on LSL–specifications. The Larch approach is called two–tiered, because it provides the shared specification language LSL and a specific interface specification language for every supported programming language (Ada, C, C++, CLU, ML, Modula–3, and Smalltalk are currently supported).

**Wide–Spectrum Languages** Whereas interface specification languages aim at bridging the gap between (existing) programming and specification languages, wide–spectrum languages are designed to provide several levels of abstraction and programming styles within one language. Ancestors of most of the wide–spectrum languages are VDM (cf. [BJ78]) and CIP (cf. [BBB+85]). VDM supports a classical model based approach to software development. CIP comprises abstraction levels as different as algebraic specifications and jumps in imperative programming. A newer development is the design language COLD–K (cf. [FJ92]). It follows the tradition of VDM with influences from ASL, dynamic logic, and object–oriented languages. COLD–K is mainly developed to specify state–based systems and only supports the specification of abstract programs.

An interesting approach to relate object–oriented specifications of abstract data types to object–oriented programs is developed in [Bre91]. In that approach, the integration of specifications and programs is achieved by giving the programming language semantics in terms of the specification language and by using so–called state based homomorphisms as abstraction functions.

**Universal Specification and Verification** Formal specification and verification of programs rely to a large extent on properties of program–independent data types and functions like integers, floating point numbers, lists, sets, streams, sorting functions, and ordering relations. The missing specification and verification support for these universal data types was one of the major problems that the developers of the Stanford Pascal Verifier had to face. The progress made in this area is certainly bigger than recognized within software development practice. Powerful universal specification languages are available (cf. e.g. [BFG+93, CEW93, GWM+92]) with sophisticated type systems and executable sublanguages for prototyping. Even more important for the vision underlying this thesis, most of the specification constructs are supported by interactive or (half–)automatic theorem provers (cf. e.g. [COR+95, PN90, GG89]). These provers especially provide powerful decision procedures for basic data types and support standard induction schemes so that most of the simple proof steps can be done by the machine.

**Program Verification** Since the seminal articles on programming logic by Floyd ([Flo67]) and Hoare ([Hoa69]), the area of program verification has seen rapid development. Hoare logic that started as a partial correctness logic for a very basic imperative programming language without procedures was extended to a total correctness logic supporting almost all language constructs existing in imperative languages, in particular block structure, procedures, data structures (including arrays and pointer structures), and a variety of control constructs (including jumps and constructs for concurrent programming). A survey of Hoare logic is contained in [Cou90].

In parallel with Hoare logic, similar and — later on — more powerful or flexible programming logics have been developed: predicate transformer logics (cf. [Gri81]); dynamic and temporal logics (in particular for the verification of parallel programs; cf. [GU91]); a logic based on predicative program specifications (cf. [Heh93]). Logics for object–oriented programs have to deal with similar problems as logics for pointer structures. We learned from [Bij89] that presents predicate transformers for this purpose. Concerning object creation, we consulted [AdB94] that deals with dynamically evolving process structures where processes are represented by objects.

**Towards the Vision** Despite the progress sketched in the previous paragraphs, it will still take several years of concentrated research and development efforts to turn the vision sketched above into everyday program development practice. The presented work should be considered as part of this effort. The taken approach, an outline of the presentation, and the major contributions are summarized in the following section.

## 1.2 Approach, Outline, Contributions

The presented work was driven by two central goals: 1. An integrated formal framework for specification and verification of object–oriented programs that can be used as a basis for interactive proof environments. 2. Interface specification techniques that support implementation–independency and are in particular applicable to realistic software libraries. To be applicable to realistic software libraries, the specification techniques have to be able to handle data structure sharing and destructive updates. This section sketches the approaches taken to achieve these goals. It provides an outline of the following chapters and summarizes the contributions of the work.

**Integrated Framework** The integration of interface specification and verification for object–oriented programs presented in this thesis is based on formal programming language definitions. Such definitions are structured into a static and a dynamic part. The static part defines in particular the sorts and functions needed to model so–called object environments (cf. [Bre91]). An object environment describes for a program state which objects are created and what their state is by mapping instance variables to values. The static part is specified in an algebraic way[3] so that standard theorem proving techniques can be applied. The dynamic part defines how the state of program variables and the object environment evolves during program execution. It is specified as an axiomatic semantics based on Hoare triples. Compared to other language formalizations, e.g. operational or denotational definitions, a Hoare–style axiomatic semantics has the advantage that interactive program verification can be

[3]This idea appears in several contexts; cf. e.g. [Bro96].

directly based on the program text and needs no intermediate semantics representation (cf. [GMP90] for a discussion of this topic). Object environments are handled in Hoare logic via a special global variable. This variable is not accessible in programs, but can be referenced in pre– and postconditions. The meaning of interface specifications is given by reducing them to Hoare triples.

**Interface Specification** An interface specification for an object–oriented program consists of (a) an invariant for each type/class and (b) a requires clause and a set of pre–postcondition–pairs for each method. It may use sorts and functions from universal specifications. Thus, it follows the two–tiered approach of the Larch family of interface languages (cf. [GH93]) where specifications consist of (a) an interface part written in an interface language tailored to the needs of the underlying programming language and (b) a programming language independent part written in a universal, declarative specification language. The differences between the Larch approach and the techniques developed in this thesis essentially result from the formal integration of interface specifications and programming languages:

1. Parts of the formal programming language definition can be used in interface specifications. This is in particularly true for aspects concerning object environments or memory models. These semantical prerequisites simplify interface languages (cf. section 4.2.1).

2. Larch interface languages only support a restricted number of specification styles. Simplifying a bit, an interface specification either directly refers to the concrete types, variables, and data components of the underlying program or it is based on implicit abstractions (see [CGR96] for a critical discussion). In the presented approach, the abstraction level of specifications can be freely chosen.

3. Verification provides flexibility in writing interface specifications, because it allows to derive interface properties from given ones and supports structuring and reformulations of interface specifications (cf. section 2.3.2 and 4.2.3).

4. In Larch interface languages, side–effects of procedures are expressed by so–called modification clauses listing those variables or locations that are possibly modified by the procedure. This tends to make interface specifications implementation–dependent and has the disadvantage that updates to recursive data structures are difficult to handle. We do not support modification clauses. Modifications are specified as relations between environments in pre– and poststates. To define such relations, the semantical framework supports program–independent predicates (cf. section 3.2).

As methodological specification support, different aspects of method specifications are analyzed: 1. The functional method behaviour, i.e. the relation between parameters and result. 2. The environmental behaviour describing for which objects the environment in the prestate is equivalent to the environment in the poststate. 3. The sharing behaviour specifying how sharing properties are affected by method execution.

**Outline and Contributions** The rest of the thesis is structured into three main chapters and a conclusion.

Chapter 2 provides an introduction to interface specifications. Throughout the chapter interfaces are considered as entities of their own, i.e. independent of any implementation. The chapter explains the differences between operational and declarative interface specification techniques. Then, focusing on declarative specifications, it discusses the syntax and different aspects of such specifications. Its last section defines subtyping of interfaces on the syntactical as well as on the behavioural level.

Chapter 3 starts at the implementation side. It defines a programming language with recursive classes and recursive methods, but without subtyping (i.e. a class–based language according to [Weg87]). Object environments are defined in an algebraic way; the dynamic semantics is given by Hoare–style axioms. The environment definition is extended by predicates expressing reachability of objects in an environment and equivalence relations on environments. A full programming logic is presented by adding language–independent axioms and rules to the axiomatic semantics. Finally, this logic is used to prove properties that hold for all programs of the language. In particular, type safety is proved; i.e. we show that poststates reached from well–typed prestate are well–typed, too.

Chapter 4 combines the interfaces of chapter 2 with the class–based language of chapter 3 and adds encapsulation and a primitive module concept. The resulting language can be considered as the kernel of most existing imperative typed object–oriented languages. The combination needs only minor changes to the definitions given in chapter 2 and 3. The programming logic is extended by an axiom and a rule. The axiom specifies the behaviour of so–called cast methods. The rule explains subtyping and allows to prove properties of abstract methods, i.e. methods that have no implementation. The programming logic is illustrated by proving type safety for the object–oriented kernel language in the sense described above. Section 4.2 investigates interface specifications supported by the developed framework. It demonstrates the relation between interface specifications and implementations. Then, it defines the formal meaning of interface specifications and analyzes related methodological issues. The final section illustrates different verification techniques.

As orientation for the reader, we summarize the main contributions of this thesis:

1. A method to formally integrate specification and verification of object–oriented programs.

2. Techniques to specify relations between object environments.

3. Techniques to specify destructive updates and sharing of data representation in an implementation–independent way.

4. Adaption and small extension of Hoare logic so that object–oriented programming languages with subtyping can be verified.

5. A formal meaning of interface specifications, in particular of class invariants.

6. An analysis how program extensions influence specified and verified programs.

## 1.3 Formal Background

The general technique underlying this thesis can be formulated in different formal frameworks. Thus, we would like to be as loose as possible about framework specific details. On the other hand, we are concerned with formal reasoning and therefore we cannot be sloppy about details. We try to cope with this conflict by

- working with standard specification techniques,
- being formal whenever program related reasoning is concerned,
- using mathematical argumentation for other proofs.

As standard specification techniques, the thesis presupposes many–sorted first–order specifications (cf. e.g. [Wir90], section 3.1 for an algebraic setting and [End72], section 4.3 for a logical setting) and recursive data type specifications (see below). This section introduces the notations used for these specifications.

A *(many–sorted) signature* $\Sigma$ is a tuple $\langle S, F\rangle$ where $S$ is a set of *sorts* and $F$ is a set of *functions* equipped with a mapping $\mathit{fsig} : F \rightarrow (S^* \times S)$. $\mathit{fsig}$ maps each function to its *(function) signature*. Function signatures are written in the form $(s_1 \times \ldots \times s_n \rightarrow s_0)$. Functions with signatures of the form $\rightarrow s_0$ are called *constants* of sort $s_0$.

In the following we assume that all many–sorted signatures contain a sort *Boolean* and for each sort $s$ the function $=_s$ with signature $(s \times s \rightarrow \mathit{Boolean})$ denoting the equality on sort $s$; we drop the subscript if sort $s$ is clear from the context.

Let $\Sigma = \langle S, F\rangle$ be a signature, $\mathit{VAR}$ be an $S$–sorted set of *logical variables* and let the set of $\Sigma$–*terms* of sort $s$, denoted by $T(\Sigma, \mathit{VAR})_s$, be defined as usual (cf. e.g. [Wir90], section 2.1). The set of $\Sigma$–*formulas* $\mathit{WFF}(\Sigma)$ is the least set satisfying the following properties:

i) every term of sort *Boolean* is in $\mathit{WFF}(\Sigma)$;

ii) if $G, H \in \mathit{WFF}(\Sigma)$, then $\neg G, (G \wedge H), (G \vee H), (G \Rightarrow H)$, and $(G \Leftrightarrow H)$ are in $\mathit{WFF}(\Sigma)$;

iii) if $X_s \in \mathit{VAR}_s$ and $G \in \mathit{WFF}(\Sigma)$, then $(\forall X_s : G), (\exists X_s : G) \in \mathit{WFF}(\Sigma)$.

$\Sigma$-formulas are denoted by bold capital letters $\mathbf{P}$, $\mathbf{Q}$, etc. To avoid parentheses, logical operators are equipped with the following precedences: "$\neg$" has highest precedence, i.e. binds stronger than any other operator; then with decreasing precedence: $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$.

Substitution of all free occurrences of a variable $X$ by a term $t$ in formula $\mathbf{P}$ is denoted by $\mathbf{P}[t/X]$. In the same way, we allow the substitution of all occurrences of a constant $c$ by a term $t$ in formula $\mathbf{P}$, denoted by $\mathbf{P}[t/c]$. In both cases, the sort of the term has to be equal to the sort of the variable or constant.

We assume the following four basic data types with the usual operations: Data type Boolean with sort *Boolean* and constants TRUE and FALSE. Data type Nat of non–negative integers with sort *Nat*, the usual operations and the canonical total ordering. Data type Integer with infinite sort *Integer* and finite sort *Int*, the usual integer operations, and two constants *minint* and *maxint* of sort *Integer*; the conversion between *Integer* and *Int* is done by the functions *intc* and *inte*:

$$\begin{array}{l} \mathit{intc} : \mathit{Integer} \ \to \ \mathit{Int} \\ \mathit{inte} : \mathit{Int} \ \to \ \mathit{Integer} \\ \quad \mathit{intc}(\mathit{inte}(F)) = F \\ \quad \mathit{minint} \leq I \leq \mathit{maxint} \ \Rightarrow \ \mathit{inte}(\mathit{intc}(I)) = I \end{array}$$

i.e. sort *Int* corresponds to the finite integer range $[\mathit{minint}, \mathit{maxint}]$.

Recursive data types play an important role in specification and programming. Accordingly, almost every specification framework and many programming languages provide a special construct for their definition (cf. e.g. [BFG+93, COR+95, Pau91, OW97]). We use the following notation to define recursive data types:

$$\begin{array}{llllllllll} \multicolumn{10}{l}{\textbf{data type}} \\ \mathit{DSrt}_1 & = & \mathit{constr}_{1,1} & (\,\mathit{sel}^1_{1,1} : & \mathit{USrt}^1_{1,1}, & \ldots, & \mathit{sel}^{m_{1,1}}_{1,1} : & \mathit{USrt}^{m_{1,1}}_{1,1} & ) \\ & | & \ldots \\ & | & \mathit{constr}_{1,c_1} & (\,\mathit{sel}^1_{1,1} : & \mathit{USrt}^1_{1,1}, & \ldots, & \mathit{sel}^{m_{1,1}}_{1,1} : & \mathit{USrt}^{m_{1,c_1}}_{1,1} & ) \\ & \ldots \\ \mathit{DSrt}_n & = & \mathit{constr}_{n,1} & (\,\mathit{sel}^1_{n,1} : & \mathit{USrt}^1_{n,1}, & \ldots, & \mathit{sel}^{m_{n,1}}_{n,1} : & \mathit{USrt}^{m_{n,1}}_{n,1} & ) \\ & | & \ldots \\ & | & \mathit{constr}_{n,c_n} & (\,\mathit{sel}^1_{n,c_n} : & \mathit{USrt}^1_{n,c_n}, & \ldots, & \mathit{sel}^{m_{n,c_n}}_{n,c_n} : & \mathit{USrt}^{m_{n,c_n}}_{n,c_n} & ) \\ \multicolumn{10}{l}{\textbf{end data type}} \end{array}$$

Such a definition introduces the sorts $\mathit{DSrt}_i$, the constructor functions $\mathit{constr}_{i,j}$, and the selector functions $\mathit{sel}^k_{i,j}$. The sorts $\mathit{USrt}^k_{i,j}$ have to be either previously defined or in $\{\mathit{DSrt}_1, \ldots, \mathit{DSrt}_n\}$. If not needed, the selector functions can be omitted.

# Chapter 2

# Specifying Interfaces

A program module essentially defines a set of types, variables, and procedures. A module may import types and procedures from other modules. Interface specifications of modules can be considered as the *contract* between module users and providers. From the user's point of view, the interface guarantees a certain module behaviour. From the provider's point of view, it formulates the requirements the implementation has to fulfill. An interface can be considered as an abstraction of a module. It specifies its behaviour as seen from the outside and hides implementation details. On the other hand, interfaces exist without referring to any implementation. In particular, different implementations can satisfy the same interface specification. In this chapter, we will stress that interfaces are entities in their own right. Essentially, there are four reasons for using interface specification in the program development process:

- Documentation.
- Implementation–independency of interfaces.
- Correctness of programs.
- As a link between programs and design specifications.

This chapter provides an *informal* introduction to interface specifications. It illustrates an important application domain of specification and verification techniques for programs. And it is meant to motivate and prepare the formal setting developed in later chapters. Section 2.1 discusses existing techniques for describing interfaces. Section 2.2 investigates declarative interface specifications in more detail and illustrates module properties that have to be captured by an interface specification in order to be applicable to software libraries of realistic languages. Section 2.3 extents this investigation to interface specifications for programming languages supporting subtyping.

## 2.1 Techniques for Interface Descriptions

A program consists of several modules declaring types, variables, and procedures together with their implementation. The interface of a module describes the syntactic and semantic module properties. The syntactic interface of a module comprises the type, variable, and procedure identifiers as well as the types of variables, procedure parameters, and procedure results. The semantic interface of a module captures the behaviour of the procedures and data structures of the module.

**Modules and Interfaces: Informal Definition** Throughout this thesis, we consider only a special kind of modules. A *module* is a collection of types. Procedures are associated with types, i.e. each procedure belongs to exactly one type. The *interface of a module* is given by the interfaces of its types. The *(syntactic) interface of a type* T consists of the type identifier and the signatures of the procedures belonging to T. The *signature of a procedure* consists of the procedure identifier, the types and identifiers of parameters, and the result type. Procedures have a restricted form: If a procedure P belongs to type T, the first parameter of P is of type T. The first parameter is always denoted by the identifier "self". As type and identifier of the first parameter are clear from the context and need no extra declaration, the parameter can be dropped in the procedure signature and kept implicit. Accordingly, the parameter self is often called the *implicit parameter*. Procedures associated with types and implicit first parameter are called *methods* in the following. A call to a method m of type T with two explicit parameters is written as

```
E0 . m( E1, E2 )
```

where E0 is an expression of type T. As subtyping is not considered in this and the following section (subtyping is introduced in section 2.3), such a method call has the semantics of an ordinary procedure call with three parameters. The only difference is that methods associated with different types may have the same identifier. This overloading is resolved by the (static[1]) type of the expression for the implicit parameter, i.e. the type of E0 in the example. This is the same mechanism as used for the selection of record components in ordinary imperative languages.

To illustrate the above explanations, we consider the syntactic interface of a list type with the following methods:

```
interface  LIST  is
   meth  empty(): SAME
   meth  isempty(): BOOL
   meth  first(): INT
   meth  rest(): SAME
   meth  append( n: INT ): SAME
   meth  updfst( n: INT )
end
```

The keyword `SAME` stands for the type identifier of the declared interface (here `LIST`). The use of `SAME` is only relevant in connection with inheritance (see 2.3.1). As shown by interface LIST there are six methods associated with type LIST. Method empty

[1]As we do not consider subtyping in this section, the static and dynamic type of an expression are identical.

and updfst deserve some explanation. Method empty creates a representation of the empty list. It does not need arguments, but according to the above restriction it has an implicit first parameter of type LIST. There are two ways to handle such cases: Either one supports a special kind of methods without implicit parameter[2] or one uses arbitrary objects of correct type as actual first parameter. We adopt the second technique in order to keep the number of method formats, and thus the number of proof rules small (cf. 3.1.1). But we support a special syntax for calling such methods so that this decision does not affect the readability of programs. This technique only works if each type T has at least one denotable object. We provide a void object for each type T, written by "void(T)", i.e. a call to empty could be written as "void(LIST).empty()". To come closer to common notation, this may be written as "LIST::empty()".

Method updfst updates the first element of a list representation. The intention is that elements in a list can be changed without having to change the representation of the list structure. According to this intention, the following program fragment exchanges the first two elements of list l without creating new objects or modifying the list structure:

```
tmp :=  l.first() ;
l.updfst( l.rest().first() ) ;
l.rest(). updfst( tmp )
```

The intention about the behaviour of methods is of course not described by the syntactic interface of a type. Only the mnemonic choice of identifiers may suggest what a method is doing. (E.g. most computer scientist would conclude that the given interface can be used to manipulate integer lists and that method rest yields the rest of a list. But of course, there are implementations for this interface showing an arbitrarily different behaviour.) In interfaces of nonstandard problems, even carefully chosen identifiers would not help. In this section, we investigate techniques for describing interfaces and method behaviour in an implementation–independent way.

**Classifying Interface Specifications** Currently, software modules are mostly documented by their syntactic interface, an informal description of their behaviour, and a couple of examples to illustrate their application. The advantages of informal descriptions are readability and flexibility. They can easily express different kinds of interface properties like partial correctness, termination, sharing, and complexity properties. Their biggest disadvantages are that they are imprecise and incomplete and that they can not be used for formal program development, automatic program analysis (cf. e.g. [Van93]) and verification.

These disadvantages have led to the development of interface specification languages, i.e. languages that allow to formulate interface properties in a formal syntax[3]. An interface specification language (IS–language for short) has to bridge the gap between the operational, state–based world of programs and the declarative, state–less

[2]Such methods are sometimes called class methods or static methods.
[3]Most of these interface languages do not have a formal semantics.

properties specifying program behaviour. Interfaces are usually specified by invariants and method annotations. Method annotations are given by pre– and postconditions. In order to classify IS-languages, it is helpful to consider two extreme approaches:

- *Operational* IS-languages where annotations are boolean expressions of the underlying programming language.
- *Declarative* IS-languages where annotations are formulas that may refer to program variables, but not to procedures/methods of the underlying programming language.

As will become clear in the following chapters, the goals of this thesis can only be reached with declarative interface specifications. Nevertheless, we briefly introduce operational interface specifications in the next paragraph to point out the differences between these two approaches and to provide some material for comparing them. The arguments brought up in this comparison hold accordingly for IS-languages that try to combine both approaches (cf. [Luc90]).

**Operational Interface Specifications** We call interface specifications *operational*[4] if the invariants and method annotations are boolean expressions of the underlying programming language (a typical example are the so–called assertions in Eiffel; cf. [Mey92]). Consequently, annotations are restricted to the data types expressible in the underlying programming language. And in general, it cannot be excluded that method calls occurring in such annotations do not terminate or modify the state. The advantages of operational interface specifications are (a) that they are easy to learn and (b) that they are executable, i.e. annotations can be checked at runtime. This way, they provide a structured testing facility that turns out to be of good practical use. On the other hand, the expressive power of such annotations is rather limited:

1. They do not enable to use functions and abstract data types that are defined in general, programming–language–independent specification frameworks.
2. They do not support the use of free variables or quantification.
3. They cannot guarantee termination and definedness of annotations (methods used in annotations may be only partially defined).
4. In general, they are unable to specify recursive procedures or updates to recursive data structures.
5. They are implementation–dependent, i.e. an interface or method cannot be specified without referring to the implementation.
6. They are not suitable for verification.

[4]The adjective "operational" should stress the fact that these annotations are subject to execution whereas declarative annotations are either valid or not w.r.t. a logical structure.

We illustrate some of these disadvantages by discussing operational method specifications for the above list interface. Preconditions are prefixed by the keyword `pre`, postconditions are prefixed by the keyword `post`. An informal, partial correctness semantics of an operational method specification could be phrased as follows: The implementation of a method m *satisfies* its annotation, iff the following holds for all possible states S in which an execution of m is sensible[5]: If execution of m's precondition in state S terminates without modifying S and evaluates to true, if execution of m in state S terminates with resulting state S′, and if execution of m's postcondition in state S′ terminates (without modifying S′), then the postcondition evaluates to `true`. As first example, we consider a specification for method `empty`:

```
meth  empty(): LIST
  pre   true
  post  result.isempty()
```

The postcondition guarantees that calling the method isempty on the result object of method empty yields true. In particular, the result of empty does not depend on the implicit parameter self. As next example, we consider a specification of isempty:

```
meth  isempty(): BOOL
  pre   true
  post  result = (self = LIST::empty())
```

This guarantees that method isempty yields true if the self–parameter equals the object produced by method empty. It should be noted here that the specification does make only sense if method empty always returns the same object. In general, this is not the case. E.g. many implementations of doubly linked lists represent the empty list by a non–void object, i.e. there are different objects representing the empty list. Thus, the specification of method isempty enforces constraints on the representations of lists. The same problem occurs if we try to specify method first in an implementation–independent way, i.e. without referring to the data components used to implement lists:

```
meth  first(): INT
   pre   not self.isempty()
   post  self = self.rest().append( result )
```

In the usual singly linked list implementation (a list object has the two attributes head and tail, method first returns attribute head, and method append returns a newly allocated list object) the implementation of method first does not satisfy the annotation, because the object returned by append is newly allocated and therefore unequal[6] to the self–object. Of course, we can get rid of the above problem, if we give up implementation–independency. Then, the specification of method first becomes almost identical to the method body:

[5]We consider a state to be *sensible* if the corresponding execution has started in a well–defined initial state.

[6]As the postcondition is a boolean expression and we assume here the semantics found in existing object–oriented languages, the equality sign denotes equality on object identities/references, not structural equality.

```
meth  first(): INT
   pre   not self.isempty()
   post  result = self.head
```

Unfortunately, the problem would return for recursive procedures. We need somehow the possibility to abstract from the representations of lists by linked objects. Essentially, there are two ways to do this: 1. Define an abstraction function taking the list representation by linked objects and yielding a value of an abstractly defined list data type (cf. the seminal paper [Hoa72] of Hoare). This cannot be done in pure operational IS-languages, because they do not provide the means for working with abstract data types. Abstraction functions are a central part of declarative interface specifications (see section 2.2). 2. Declare a set of additional observer methods, i.e. methods that "observe" the properties of interest of the underlying data structures without modifying the state. E.g. for lists we can use the following observer methods:

- `l.length()` returning the length of list `l`.
- `l.elem(i)` returning the i-th element of list `l`.
- `l.wf()` testing whether the list representation of `l` is well formed (e.g. a singly linked list representation is well formed if it is acyclic).
- `l1.disj(l2:LIST)` testing whether the representations of `l1` and `l2` are disjoint, i.e. do not share parts of their representation.

Such observer methods allow to formulate many interesting method properties, in particular requirements that must hold in the prestate of method executions. As an example, we specify the functional behaviour and the preconditions for list concatenation that may perform destructive updates on one of the list parameters (e.g. in a singly list implementation by updating the tail attribute in the last object of the self–parameter):

```
meth  concat( l: LIST ): LIST
   pre   self.wf()  and  l.wf()  and  self.disj(l)
   post      for all i in [ 1 .. initial(self.length()) ] :
               initial( self.elem(i) ) = result.elem(i)
        and  for all i in [ 1 .. initial( l.length()) ] :
               initial( l.elem(i) )
                   = result.elem(i+initial(self.length()))
```

The specification requires that both parameters are well–formed and that they are disjoint. (Otherwise destructive updates may lead to non–well–formed list representations.) The above specification does not only show the use of observer methods, but illustrates as well some additional language constructs that are often found in operational IS-languages. The `for all` construct enables universal quantification over finite domains. The `initial` construct allows to take the value of an expression in the prestate of the method execution. As we wanted to allow destructive changes on the parameters, it is mandatory to use the `initial` construct in the above specification.

Even if observer methods provide some degree of abstractness, they are fairly hard to use in more realistic examples.

The rest of this thesis is concerned with declarative interface specifications. Declarative interface specifications overcome all mentioned disadvantages of operational interface specifications. On the other hand, declarative interface specifications are in general not executable and special implementation effort is needed to generate runtime checks from annotations where this is possible.

## 2.2 Declarative Interface Specifications

In declarative interface specifications, invariants, pre-, and postconditions are given by formulas of declarative specification frameworks. These formulas may refer to program states through the program variables or by selecting object attributes. This section introduces the general concepts underlying most declarative interface specification languages. The formal meaning of declarative interface specifications, in particular the interplay between invariants and method annotations, is presented and discussed in section 4.2.2.

### 2.2.1 Introduction to Declarative Interface Specifications

**Formulating Abstraction** The user of an interface is essentially interested in *what* service is provided by the interface and not *how* this service is achieved by the implementation. The basic idea of *declarative interface specifications* is to explain the service provided by an interface on an abstract, implementation–independent level. Then, the vocabulary of the *abstract level* can be used to specify the interface behaviour. Again, we consider the list interface of the previous section as example. The abstract level for this example that has to be explained to the user is the abstract data type of integer lists with the following functions:

$$\begin{array}{lcl}
empt & : & \rightarrow\ List \\
isempt & : & List\ \rightarrow\ Boolean \\
fst & : & List\ \rightarrow\ Integer \\
rst & : & List\ \rightarrow\ List \\
app & : & Integer \times List\ \rightarrow\ List \\
conc & : & List \times List\ \rightarrow\ List \\
lng & : & List\ \rightarrow\ Integer
\end{array}$$

On the implementation or *representation level*[7], lists are represented by sets of linked objects. The connection between representation and abstract data type is expressed by abstraction functions. Abstraction functions map the concrete data types of the programming language to the sorts of the abstract data types. To be more precise, we

[7]We prefer the name *representation level*, as the representation level may still correspond to several concrete implementations.

assume in the following that for each type of the programming language there exists a corresponding abstraction function. For booleans, the abstraction[8] is trivial, i.e. a bijection $aB$ from the concrete type BOOL to the abstract sort *Boolean*. For integers, it is an injection $aI$ from the finite type INT of integer objects into the infinite sort *Integer*. For lists, it maps pairs consisting of a list object and the current environment of all other objects to values of sort *List*. I.e. if we denote the abstraction function for lists by $aL$, we get the following function signature[9]:

$$aL : \text{LIST} \times \mathit{ObjEnv} \;\rightarrow\; \mathit{List}$$

That is, in general the abstraction function takes two arguments the second of which is the object environment capturing the links between objects. The current object environment can be considered as a global program variable of sort *ObjEnv*. It is denoted by \$ in the following, i.e. $aL(\text{l}, \$)$ denotes the abstraction of list l in the current object environment.

Abstraction is such an important operation that we introduce an abbreviated notation for it. If vp is a variable or parameter of the standard types BOOL or INT, then vp! denotes $aB(\text{vp})$ or $aI(\text{vp})$ respectively. If vp is variable or parameter of a user declared type T and $aT$ denotes the abstraction function for type T, then vp! denotes $aT(\text{vp}, \$)$. I.e. we use "!" as an overloaded postfix operator where overloading is resolved by the declared type of the variable or parameter. This operator is called the *abstraction–operator*.

**Formulating Interface Specifications** From an abstract point of view, the behaviour of a method can be specified as a relation of the parameters, the object environment before executing the method body, the result, and the object environment after executing the body. From a methodological point of view, it is helpful to structure this relation into different aspects. We distinguish three different aspects. If we talk about the *functional behaviour* of a method, we refer to the relation between the abstract values represented by the parameters in the environment before method execution and the abstract value represented by the result in the environment after method execution. If we talk about the *environmental behaviour* or the *invariance properties* of a method, we refer to properties relating the object environment before to that after method execution. If we talk about the *sharing behaviour* or *sharing properties* of a method, we refer to properties relating the representations of parameters and result.

The specification of the functional behaviour of the list methods essentially reduces to the requirement that abstraction satisfies a homomorphism–criterion: Applying the abstraction to the method parameters and then using the list function on the abstract level yields the same value as executing the method and applying the abstraction to the result. For the methods empty and isempty, this leads to the following specifications:

[8] Abstraction on Booleans is only used for technical reasons that become clear in chapter 3.

[9] Considering this chapter as informal introduction, we use program types, e.g. LIST, in signatures; in chapter 3, the types are replaced by the corresponding sort for objects.

```
meth  empty(): LIST
   post  result! = empt

meth  isempty(): BOOL
   pre   self! = SL
   post  result! = isempt(SL)
```

Trivial preconditions, i.e. preconditions that are tautologies, can be omitted as shown for empty. In the specification of isempty, the free variable $SL$ is used to capture the abstraction of self in the prestate so that it can be used in the postcondition. In the operational interface specification we used the `initial` construct to express evaluation of an expression in the prestate (see end of section 2.1). Here, we denote the corresponding operation for declarative interface specifications by the postfix operator "`^`" (following the notation of the Larch interface language for C, called LCL; cf. [GH93]). This operator is called the *prestate–operator*. The prestate–operator may be applied to terms in postconditions that do not already contain an occurrence of it or of the result variable. Such terms are called *prestate–terms*.

The value of a term depends on the program variables, parameters, and the current environment variable $. A state assigns values to program variables, parameters, and the environment variable. Thus, the interpretation of a term is state–dependent. The prestate–operator interpretes a term according to the prestate of a method execution. This way, the pre– and poststates can be related in one formula. E.g. the following specification of method isempty is equivalent to the one above:

```
meth  isempty(): BOOL
   post  result! = isempt( self!^ )
```

To understand why the prestate–operator cannot be omitted without changing the meaning of the specification, it is helpful to eliminate the abbreviating abstraction–operator:

```
meth  isempty(): BOOL
   post  aB(result) = isempt( aL(self,$)^ )
```

The specification makes no statement about the environmental behaviour of method isempty; in particular, it may be possible according to the above specification that isempty causes side–effects on the environment. If this were the case, the interpretations of $aL(\mathrm{self}, \$)$ in pre– and poststates could be different.

The functional behaviour of methods first, rest, and append is specified in the same way. The prefix operator "`~`" is used as negation sign in typewriter font[10]:

[10]The reason to use typewriter font instead of nicely layouted formulas to denote pre– and postconditions of methods is that we want to distinguish between the bare syntax in which a user writes down a specification (typewriter font) and the mathematical meaning of such specifications (layouted formulas) being explained in section 4.2.2.

```
meth  first(): INT
   pre  ~isempt( self! )
   post  result! = fst( self!^ )

meth  rest(): LIST
   pre  ~isempt( self! )
   post  result! = rst( self!^ )

meth  append( n: INT ): LIST
   post  result! = app( n!^, self!^ )
```

The illustrated specification technique works very well for *applicative* methods, i.e. for methods that do not produce side–effects or perform destructive updates and return a result. And of course, verification of applicative methods is simpler than the more general case. Unfortunately, considering only applicative procedures/methods would severely restrict the class of programs that can be specified: Almost all software libraries (e.g. standard template library, Leda), tool kits (e.g. XToolKit), or software interfaces (e.g. system calls in Unix, interfaces to data bases) enable the user to produce side–effects and perform destructive updates. To be applicable to such programs as well, interface specifications have to provide information telling the user which parts of the object environment are possibly modified by a method and which parts remain invariant under a method execution. Thus, an interface specification has to describe not only the functional behaviour of methods, but as well certain aspects from the representation level. The next section shows how such aspects can be expressed in an implementation–independent way.

#### 2.2.1.1 Invariance and Sharing Properties

Many interface specification languages (cf. e.g. [GH91], [FJ92]) use so–called modify clauses to describe which variables and record fields are modified by a procedure. A modify clause is essentially a list of variable names or expressions yielding L–values or – in semantics terms – locations. Such specifications have essentially three disadvantages, in particularly in connection with complex data types:

1. It is difficult to describe modifications in linked object structures, e.g. to describe that the last record in a singly linked list is updated.

2. It tends to be implementation–dependent, because the locations that are modified have to be known to the reader of the specification.

3. It is not appropriate for verification, i.e. there is no programming logic directly supporting modifies clauses; for verification, modifies clauses have to be translated into statements of the form: If L is a location not mentioned in the modifies clause, then the value hold by L in the prestate is the same as the value of L in the poststate.

The second aspect is even worse in cases where an interface is implemented in different ways. E.g. the interface LIST can be implemented by singly linked objects or by using arrays (these are the two implementations we use as examples in chapter 3 and 4). Both implementations may show the same abstract behaviour; but as they have different locations, the behaviour had to be described for each implementation differently. The Larch interface languages[11] solve this problem by supporting only coarse grained modification specifications: It can be specified that the representation of an abstract value (e.g. a linked object structure) is modified or is left unchanged. It cannot be specified that only parts of a representation are modified (cf. [CGR96, PH95] for a discussion).

Instead of describing what is modified, we support techniques to specify sharing properties in an abstract manner and to specify invariance properties based on the object environment. In this subsection, we give a short introduction to these techniques. The formal background and the relation of these techniques to program verification is presented in the subsequent chapters.

**Invariance Properties** As our example language does not provide global variables, side–effects and modifications performed by a method are always related to the object environment. In order to express properties on the object environment, we use the symbol $ to denote the current environment. $ should be regarded as a global variable of sort *ObjEnv*. Leaving the object environment unchanged means that $ has the same value in the prestate as in the poststate. For example method first shows such a behaviour:

```
meth  first(): INT
   pre  ~isempt( self! )
   post  $^ = $
```

Now, the specification of method first consists of two pre–post–pairs: One describes the functional behaviour and one describes an *invariance property*. Before continuing the specification of invariance properties, we need a closer look at the object environment. In our approach, the object environment captures all state–dependent information about objects, i.e. how objects are linked and whether an object is alive[12] in the current state. We call an object *alive* in a state $E$ if it was created/allocated in the execution leading to $E$. Thus, the above specification of method first guarantees in particular that method first does not create objects.

Specifying invariance properties means to specify relations on object environments. An important relation on object environments is $X$–*equivalence*: Two environments $E$ and $E'$ are called equivalent w.r.t. an object $X$ denoted by $E \equiv_X E'$ if

- $X$ is alive in $E$ if and only if it is alive in $E'$, and
- if $X$ and all objects reachable from $X$ have the same state in $E$ and $E'$.

[11] We simplify here a bit; in particular, the Larch interface language for C++ (cf. [CL94]) is more expressive.

[12] Deletion or killing of objects is not considered in this thesis.

A formal definition of $X$–equivalence is given in chapter 3. $X$–equivalence allows us to specify that an object environment is left invariant in all aspects relevant to an object or a set of objects. A well–known property of this kind is the absence of side–effects: A method is *free of side–effects*, if it leaves the object environment invariant for all objects that are alive in the prestate. For example the methods append, rest, and empty are free of side–effects. Here is the specification for method append[13]:

```
meth  append( n: INT ): LIST
   pre   alive(X,$)
   post  $^ ≡_X $
```

Using the same technique, we can specify that a method leaves the environment invariant for all objects of certain types.

As a more interesting example, we study method updfst. Method updfst is used here to illustrate the specification of methods without result, of sharing properties, and of destructive updates. Method updfst is assumed to update the first element of a list. Thus, abstracting the self–parameter in the poststate yields the same result as abstracting the self–parameter in the prestate, taking the rest of it, and appending the integer parameter:

```
meth  updfst( n: INT )  is
   pre   ~isempt( self! )
   post  self! = app( n!^, rst(self!^) )
```

Although essentially a specification of the functional behaviour of updfst, this specification implicitly tells us that the self–object is modified: Abstracting self in the prestate yields in general a different result from abstracting self in the poststate. But knowing only that method updfst produces some side–effect is too vague. In particular for verification, we need to know

1. which list representations are affected by a call to updfst and
2. what precisely is the effect.

A list representation usually consists of a set of linked objects. We say that two representations $XL$ and $YL$ are *disjoint* in an environment $E$, denoted by $disj(XL, YL, E)$, if the corresponding set of objects is disjoint. More general: We say that two objects are disjoint if the corresponding sets of reachable objects are disjoint. Like the predicate *alive*, the predicate *disj* is a concept of the object environment. It is based on the data and state model of the programming language and works in all programs. E.g. in chapter 3, we prove that methods do not affect living objects that are disjoint from all their parameters (cf. lemma 3.13 and the succeeding discussion). Applied to method updfst this lemma yields:

```
meth  updfst( n: INT )  is
   pre   ~isempt( self! ) /\ alive(X,$) /\ disj(X,self,$)
   post  $^ ≡_X $
```

Before we attack the problem of specifying sharing properties, we discuss some syntactical aspects of specifying methods.

[13]Chapter 3 introduces an easier to use notation for the absence of side–effects.

**Form of Method Specifications** Up to this point of the introduction to declarative interface specifications, methods were specified by one or several pre–postcondition–pairs (for methods first and updfst, we gave two pairs). In this paragraph, we introduce a normalform of method specifications and discuss syntactical aspects concerning the structuring of specifications.

Pre– and postconditions are sorted first–order formulas (cf. section 1.3) with the restriction that (a) quantification over program variables is not allowed and (b) the variable "result" and the prestate–operator must not occur in the precondition. As assignment to parameters will be forbidden (cf. section 3.1.1) so that the object denoted by a formal parameter in the prestate is the same as in the poststate, method specifications can always be transformed into an equivalent *normalform* with the following properties:

1. The method specification consists of exactly one pre–post–pair.
2. The precondition of this pair is trivial, i.e. the constant TRUE.
3. The prestate–operator is only applied to method parameters and the environment variable (and not to compound terms).
4. The prestate–operator is applied to every parameter occurrence[14].

We demonstrate this transformation with the following example:

```
meth  m( p1,..,pn ): T  is
  pre   Q1(self,p1,..,pn,$)
  post  R1(self,p1,..,pn,result,$)

  pre   Q2(self,p1,..,pn,$)
  post  R2(self,p1,..,pn,result,$)
```

To establish the third property, we replace prestate–terms $t(\mathrm{self}, \mathrm{p}_1, \ldots, \mathrm{p}_n, \$)\hat{}$ that occurr in **R1** and **R2** (where $\mathrm{p}_i$ are the parameter occurrences in $t$) by $t(\mathrm{self}\hat{}, \mathrm{p}_0\hat{}, \ldots, \mathrm{p}_n\hat{}, \$\hat{})$. To establish the fourth property, we apply the prestate–operator to all parameter occurrences in postconditions that are still without such an application. This step does not change the semantics of the specification, because parameters hold the same object in pre– and poststates. The application of these two transformation steps to postconditions **R1** and **R2** yield two new postconditions that we denote by **NR1** and **NR2**. The third transformation step shifts the preconditions as a premise into the postcondition by applying prestate–operators to all parameters and the environment variable:

```
meth  m( p1,..,pn ): T  is
  post  Q1(self^,p1^,..,pn^,$^) => NR1(self^,p1^,..,pn^,$^,result,$)
  post  Q2(self^,p1^,..,pn^,$^) => NR2(self^,p1^,..,pn^,$^,result,$)
```

[14]This restriction simplifies syntactical transformations discussed later in this thesis.

To establish the first property, take the conjunction of all postconditions. The resulting form is close to predicative specifications according to Hehner (cf. e.g. [Heh93]). Whereas he presents a special theory for such predicative specifications, our specifications will be embedded into Hoare logic[15] (cf. section 4.2.2).

Method specifications are usually loose; i.e. they express some method properties, but may leave room for different implementations. In particular, invariance and sharing properties are often loosely specified. Just as abstract data type specifications are structured into several axioms (although they could be given by the conjunction of these axioms), method specifications can be structured into several pre–postcondition-pairs[16]. Thus, a method specification is a set of pre–post–pairs[17]. As a mandatory part, a method specification has to provide an *application requirement* on parameters and the environment in the prestate. If this requirement is met, it is guaranteed that the method does not cause a runtime error (arithmetic overflow, dereferencing of void objects). E.g. the application requirement of method updfst is `~isempt(self!)`.

The application requirement could be formulated as a pre–post–pair where the postcondition is trivial, i.e. the boolean constant TRUE. But then the requirement had to be repeated in all the other pre–post–pairs of the method specification. To avoid such a repetition, we use a special syntactic construct, the *requires clause*, to state the application requirement. The requires clause is an implicit conjunct of the preconditions of all pre–post–pairs belonging to a method specification. It is prefixed by the keyword `req`. Using this syntactic abbreviation, the so far discussed properties of updfst can be stated as follows:

```
meth  updfst( n: INT )  is
   req   ~isempt( self! )

   post  self! = app( n!, rst(self!^) )

   pre   alive(X,$) /\ disj(X,self,$)
   post  $^ ≡_X $
```

**Sharing Properties** If two data representations are not disjoint, they *share* objects. Sharing is usually done for efficiency reasons. It causes no problems as long as destructive updates are not possible. However, performing a destructive update in one representation R potentially affects all representations shared with R. Unfortunately, destructive updates in shared data representations are often desirable and sometimes even indispensable. E.g. if graphs are implemented by two dimensional

[15]This design decision is based on pragmatic reasons. Verification in Hoare logic can be arranged in proof outlines, i.e. by adding formal comments to programs. This has proved to be a good technique for interactive proof environments (cf. [GMP90]).

[16]Beside enhancing readability the structuring is needed to give invariants a formal meaning; cf. section 4.2.2.

[17]We keep the precondition although it can be eliminated by the technique demonstrated above, because the structuring into pre– and postconditions often can enhance the readability of a method specification.

arrays, insertion and deletion of edges is very efficient as long as we implement them as updates. Other examples are interfaces to file systems or data bases. In deed, most available software libraries for imperative or object–oriented programming languages work with sharing and destructive updates. Thereby, interfaces transfer some of the responsibility of managing shared representations to the users. In order to meet this responsibility, at least some aspects of how sharing is done has to be visible. Consequently, interface specifications have to be able to capture these properties.

Usually sharing of data representations follows clear patterns that can be described in an abstract implementation–independent way. To illustrate this, we consider different implementations of lists that exploit sharing: implementations by singly linked objects, by doubly linked objects, and by arrays. With all these implementations we can realize the pattern that a list representation is a direct part of another list representation. The following figure sketches this situation for the singly and doubly linked implementations. In both cases, *XL* is a direct part of *YL*.

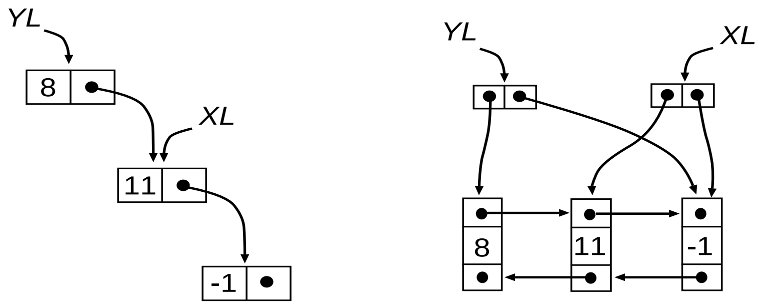


Figure 2.1: Sharing in singly and doubly linked lists

Just as we have specified the functional behaviour of a method in an abstract way, we can specify the sharing behaviour of a method in an abstract way. Instead of abstraction functions, the sharing behaviour is described with functions and predicates relating representations. For our list example, we only need the predicate *dirpart* expressing that one list representation is a direct part of another list representation in a given object environment:

$$\mathit{dirpart} : \mathrm{LIST} \times \mathrm{LIST} \times \mathit{ObjEnv} \;\rightarrow\; \mathit{Boolean}$$

Similar to abstraction functions, the specification of sharing predicates is implementation–dependent and not part of the interface specification (cf. subsection 4.2.4.3 for the specification of *dirpart*). E.g. the specification of predicate *dirpart* for a singly linked list implementation is different from that for a doubly linked list implementation. These specifications are needed for the verification of implementations. The user of an interface must not know them. For him or her, the properties of the sharing predicates (and the abstraction functions) are expressed by the method specifications, invariants (see next paragraph), and possibly some additional axioms. These additional axioms typically link properties of sharing predicates and abstraction functions. As an example, we consider the following property of *dirpart*: If *XL* is a direct part of

*YL* then the abstraction of *XL* equals the rest of the abstraction of *YL* (cf. subsection 4.2.4.3 for the formal context in which the implication has to hold):

$$\mathit{dirpart}(XL, YL, E) \;\Rightarrow\; aL(XL, E) = \mathit{rst}(aL(YL, E)) \tag{2.1}$$

where *aL* denotes the abstraction on lists (cf. section 2.2.1). Based on predicate *dirpart*, we can define a predicate *part*, the reflexive, transitive closure of *dirpart*:

$$\begin{array}{l} \mathit{part} : \mathrm{LIST} \times \mathrm{LIST} \times \mathit{ObjEnv} \;\rightarrow\; \mathit{Boolean} \\ \quad \mathit{part}(XL, YL, E) \;\Leftrightarrow \\ \qquad XL = YL \vee \exists ZL : \mathit{dirpart}(XL, ZL, E) \wedge \mathit{part}(ZL, YL, E) \end{array}$$

These definitions are sufficient to specify the sharing properties of the list interface. Based on these specifications, users of this interface can verify their programs without knowing the underlying list implementation or the precise sharing mechanism underlying the predicate *dirpart* (cf. section 4.3.3 for an example of such a proof). Specification of the sharing behaviour of methods has three aspects: 1. Establishing sharing properties. 2. Invariance of sharing properties. 3. Usage of sharing properties. For the list interface, we assume that method rest establishes the sharing property *dirpart* between its argument and result (the requires clause is omitted):

```
meth  rest(): LIST
   post  dirpart(result,self,$)
```

For method append we could specify as well that the self–object is a direct part of the result. But, this would not be valid in list implementations with arrays, because in such implementations the array has to be copied if the length of the list exceeds the size of the array. On the other hand, we like to guarantee that the result of append is never a part of some list being alive in the prestate:

```
meth  append( n: INT ): LIST
   pre   alive(XL,$)
   post  ~part(result,XL,$)
```

Because all methods of interface list except updfst are free of side–effects, we can prove that sharing property *dirpart* is left invariant for living objects under the execution of these methods. For method updfst, we have to specify the invariance explicitly:

```
meth  updfst( n: INT )  is
   pre   dirpart(X,Y,$)
   post  dirpart(X,Y,$)
```

It remains to show how sharing properties can be exploited for the specification of method behaviour. For method updfst we consider the cases that the self–object is part of a list *XL* or is not. If it is not, then method updfst does not modify *XL*, i.e.:

```
meth  updfst( n: INT )  is
   pre   ~part(self,XL,$)
   post  XL!^ = XL!
```

The most interesting case is that the self–object is part of a list *XL*. Then the side–effect on self will effect *XL* as well, i.e. the element of *XL* that corresponds to the first element of updfst is modified accordingly. This can be expressed as follows:

```
meth  updfst( n: INT )  is
   pre   part(self,XL,$) /\ XL! = conc( PREFIX, self! )
   post  XL! = conc( PREFIX , app( n!, rst(self!^) )
```

It should be noted that the second conjunct in the precondition does not strengthen the precondition, because the existence of list *PREFIX* is implied by the other conjunct. Furthermore, we like to remark that the functional behaviour of method rest and method updfst can be derived from the sharing specification together with property 2.1.

**Interface Invariants** Objects of a type usually maintain certain properties during their lifetime. Many of these properties are well–formedness conditions on data representations. E.g. in an implementation of a bounded stack the index of the top element has to be within the bounds. Another example is the requirement that singly linked lists have to be acyclic. This kind of invariants concern the representation of one abstract value. Other invariant properties may relate two or more representations. E.g. in our list interface it is the case that two well–formed list objects are either disjoint or there is a list that is part of both of them.

Such properties can be specified as *interface invariants*. An interface invariant is a environment–dependent formula with one free variable ranging over the type of the interface. Informally, an implementation satisfies an interface invariant $INV(Y,\$)$ for a type T if $INV(Y,\$)$ is valid for all living objects $Y$ of T in all object environments outside method executions of T. Syntacticly, an interface invariant is specified together with the interface. As an example, we specify that list objects are well-formed, captured by an abstract predicate *wfL*, and that the above sharing invariant holds:

```
interface  LIST  is
   inv Y:  wfL(Y,$) /\
           ALL Z: wfL(Z,$) =>   disj(Y,Z,$)
                              \/ EX W: part(W,Y,$) /\ part(W,Z,$)
     ....
end
```

The formal semantics of interface invariants, their application in verifications and the specification of implementation–dependent predicates like *wfL* is treated in the following chapters.

### 2.2.2 A Systematic View to Interface Specification

This section summarizes and clarifies the concepts introduced in the last section. In particular, it shows how the interface specifications discussed in the last section can be reduced to a more basic form. This basic form is formalized in chapters 3 and 4. An *interface specification of a type* consists of the following components:

1. The syntactic interface given by the type identifier and the method signatures; if supported by the programming language, subtyping information belongs as well to the syntactic interface (cf. section 2.3).

2. The interface invariant.

3. One requires clause and a set of pre–post–pairs for each method (cf. subsection 2.2.1.1 for a discussion on structuring method specifications into several pre-post-pairs).

4. Abstract data types modelling the abstract level used to specify the interface behaviour.

5. Signatures of abstraction and sharing functions.

6. Properties of abstraction and sharing functions described by axioms.

Interface specifications for different types may use the same abstract data types. They may even use the same abstraction operations, in particular in the presence of subtyping. Thus, if we talk of a *set of interface specifications*, we mean that each element of the set consists of one syntactic interface, one interface invariant, and for each method one requires clause and a set of pre–post–pairs. The abstract data types, signatures of abstraction operations, and their properties are not associated with specific interfaces, but considered to be global defintions.

Most of the interface components are already illustrated by the interface LIST in the last section. Here, we revisit the interface components in turn to provide syntactical clarifications and contextual constraints.

**Syntactic Interfaces, Interface Invariants, Method Specifications** The syntactic interface of a type consists of the type identifier and a possibly empty set of method signatures. The context–free syntax should be clear from the list example on page 12; several explicit parameters are seperated by commas. Methods belonging to one interface must have different identifiers. Parameters belonging to one method must have different identifiers. The identifiers "result" and "self" may not be used for types, methods, or explicit parameters. In a set of syntactic interfaces the sets of type identifiers, method identifiers, and parameter identifiers have to be disjoint. For each occurring type identifier there has to be at most one syntactic interface. A set of syntactic interfaces is called *name–closed* if there is exactly one syntactic interface for each occurring type identifier.

Interface invariants are sorted first–order formulas (cf. section 4.2.2) with one free variable. The precise signature of these formulas will be defined in the following chapters. In particular, they may contain the symbol $ denoting the current object environment, functions defined on the object environment (like *alive*, *disj*), functions from the abstract data types mentioned under point 4 above, and abstraction operations (like *dirpart*). They must not contain parameter or method identifiers or the identifier "result".

A method specification consists of one requires clause and a set of pre–post–pairs. In the formal framework used in the following chapters, requires clauses, preconditions, and postconditions are sorted first–order formulas. Beside the elements mentioned for invariants above, requires clauses and preconditions may contain the parameters of the method they are associated with. Postconditions may contain applications of the prestate–operator and the result variable if the corresponding method returns a result; method parameters may only occur within prestate–terms. The method specifications given in section 2.2.1 violate these context conditions in two ways: 1. They use the "!"–symbol to denote abstraction. 2. They use parameters in postconditions not occurring in prestate–terms. These "violations" can be considered as abbreviations and eliminated by syntactical transformation. Ad 1: The abstraction operator "!" may be only applied to terms the type of which is clear from the context. Based on this type information the abstraction operator can be replaced by an application of the corresponding abstraction function; if abstraction depends on the object environment, use the global variable $ (such replacements were illustrated together with the specification of method isempty in section 2.2.1). Ad 2: Parameters p in postconditions that do not occur in prestate–terms are replaced by p^. For verification with Hoare logic the prestate–operator has to be eliminated as well. This can be done by replacing prestate–terms $t\hat{}$ by a fresh variable, say $V$, that does not occur in the corresponding pre–post–pair and by adding "$V = t$" as conjunct to the precondition.

**Abstract Level** An interface specification uses program independent specifications of sorts and functions like the abstract data type of lists sketched in section 2.2.1. It makes use of the abstraction operations and their properties (e.g. the abstraction function *aL* for lists or the sharing predicate *dirpart* with property 2.1). Abstraction operations are based on a formal model of the object environment. Sorts and functions from the object environment can be as well used in the interface specifications (e.g. the sort *ObjEnv* and the predicates *alive* and *disj*). The technical details of these specifications depend on the used specification framework. There are quite a number of reasonable choices (cf. the discussion in section 1.3). Within this thesis, we use many–sorted first–order specifications and a construct to define recursive data types as specification framework. Both concepts are briefly introduced in section 1.3. They are sufficiently expressive for the purposes here and known by most computer scientists involved in formal methods.

## 2.3 Interface Specifications and Subtyping

If a type S is declared to be a subtype of a type T, denoted by $S \preceq T$, objects of type S can to be used in any context where objects of type T are allowed. This general concept underlying subtyping entails requirements for the type interfaces. This section investigates the resulting constraints for syntactic interfaces and discusses consequences for interface specifications.

### 2.3.1 Syntactic Subtyping

This section investigates the syntactical aspects related to adding subtyping to interface specifications. In contrast to many object–oriented programming languages, we consider subtyping here seperated from code inheritance. We assume a name–closed[18] set of syntactic interfaces containing in particular interfaces for types S, SP, SR, and T, TP, TR. To discuss the syntactic constraints on interfaces caused by subtyping, we consider the following method call statement

```
  v :=  w . m( x )
```

where w is a variable of type T, x is variable of type TP, v is a variable of type TR, and m is a method associated with type T with signature `meth m( p: TP ): TR`. Thus, the above statement is well–typed. If S is a subtype of T, S–objects can be used in any context where T–objects are allowed. In particular, variable w may hold S–objects. Essentially, there are two semantics to execute method calls in the presence of subtyping:

- Static selection: Select the method to be executed according to the type of variable w; this type is given by the declaration of w.
- Dynamic selection: Select the method to be executed according to the type of the object hold by w at call time; e.g. if w holds an S–object, execute method m associated with S.

Static selection is the semantics used in procedural languages[19]. It can be implemented more efficiently and makes subtyping simpler. Dynamic selection is the semantics used in object–oriented programming languages. It allows to specialize methods w.r.t. the objects they work on, simplifies encapsulation of implementation details, and supports interfaces without implementation (often called abstract interfaces or abstract classes). We investigate here dynamic selection, i.e. the object–oriented semantics.

Dynamic selection enforces certain constraints on the interfaces. If S is a subtype of T, the interface of S has to contain a method m for each method m associated with T. We denote these two methods by S::m and T::m. As S::m may be called in contexts where T::m is allowed, it has to be ensured that S::m and T::m have the same number of parameters and either produce both a result or not. And the types of parameters and results have to be compatible in the following sense: If TP is the type of a parameter of T::m, S::m has to handle such parameters, i.e. the corresponding parameter type SP of S::m has to be wider than TP, i.e. $TP \preceq SP$. If T::m produces results of type TR, S::m has to produce results that are acceptable in all context where TR–objects are allowed, i.e. the result type SR of S::m has to be a subtype of TR. These constraints are summarized in the following definition:

**Definition 2.1 :** Subtype Relation
A partial ordering $\preceq$ on a name–closed set of interfaces is called a *subtype relation*

[18]Cf. section 2.2.2 for the definition of “name–closed”.
[19]Several procedural languages support subtyping at least on predeclared types.

iff the following holds for all S,T with S $\preceq$ T: If T has a method m with (explicit) parameter types $T_1, \ldots, T_z$ and result type TR, then S has a method m with (explicit) parameter types $S_1, \ldots, S_z$ and result type SR and $S_j \succeq T_j$ and SR $\preceq$ TR.

□

If S $\preceq$ T, we call S a *subtype* of T and T a *supertype* of S. If S is a proper subtype of T, i.e. S $\preceq$ T and S $\neq$ T, and there is no type between S and T, i.e. S $\preceq$ ST $\preceq$ T implies S = ST or ST = T, then S is called a *direct subtype* of T and T a *direct supertype* of S. It should be clear from the above definition that a type can have more than one direct supertype, i.e. our interface specification language supports multiple subtyping. Furthermore, we like to stress that subtyping is in general a global relation on a set of types and not a local relation between two types. In particular, a set $\mathcal{T}$ of types may possess different subtype relations $\preceq_1$ and $\preceq_2$ such that T1 $\prec_1$ T2 and T2 $\prec_2$ T1 for T1, T2 $\in \mathcal{T}$ (as trivial example, consider interfaces without methods).

The subtype relation is declared together with the interfaces. Each interface declares its direct supertypes (if any). The declared subtype relation is the reflexive transitive closure of these subtype–supertype–pairs. We call a set of interfaces with subtype declarations *well–formed* if it is name–closed (in particular there exists an interface for all types mentioned as supertypes) and if the declared subtype relation is a subtype relation according to definition 2.1. For the second condition, it has to be checked that the declared subtype relation is acyclic and that the method constraints hold for subtype–supertype–pairs. The concrete syntax of supertype declarations is illustrated by the following interface CLIST that is a subtype of interface LIST (cf. section 2.1):

```
interface  CLIST  subtype of LIST  is
   meth  empty(): CLIST
   meth  isempty(): BOOL
   meth  first(): INT
   meth  rest(): CLIST
   meth  append( n: INT ): CLIST
   meth  updfst( n: INT )

   meth  concat( l: CLIST ): CLIST
   meth  copy(): CLIST
 end
```

In addition to the functionality of type LIST, type CLIST provides a method for concatenating lists and a method for copying lists. To avoid rewriting the method signatures already contained in LIST, we use the following equivalent syntax:

```
interface  CLIST  subtype of LIST  is
   include LIST
   meth  concat( l: CLIST ): CLIST
   meth  copy(): CLIST
 end
```

The `include` directive includes the method signatures from interface LIST. The two interfaces for CLIST given above are equivalent, because we used the type placeholder `SAME` in the declaration of LIST: Whereas in interface LIST it denotes type LIST, in interface CLIST it denotes type CLIST.

As another subtype of LIST, we consider the type PLIST. PLIST is used in the following as a tiny example to illustrate data types with internal state and with operations that work with side–effects. PLIST provides methods to iterate over lists and read (get) and write (set) the current list position. Method init initializes the position to the first element of the list, method isdef tests whether the current position is defined, and method next moves the current position to the next element of the list:

```
interface  PLIST   subtype of LIST  is
   include LIST
   meth  init()
   meth  isdef(): BOOL
   meth  next()
   meth  get(): INT
   meth  set( n: INT )
end
```

It is easy to check that interfaces CLIST and PLIST satisfy the subtype constraints of definition 2.1.

### 2.3.2 Semantic Subtyping

The definition of syntactic subtyping guarantees the existence of consistently typed methods in subtypes. It makes no statement about the semantical relation between a method in a subtype and the corresponding method in the supertype. *Semantic subtyping* means that methods in the subtype behave according to the specification of the corresponding methods in the supertype. The framework presented in this thesis can be used to give semantic subtyping a precise meaning, to analyze shortcomings and relate semantic subtyping to verification. This subsection discusses problems with existing definitions of semantic subtyping and illustrates them with the list interfaces from the previous sections.

In most frameworks, an interface specification for a type consists of an interface invariant and one pre–post–pair for each method (we demonstrated in section 2.2 how our interface specifications can be translated into this format). Semantic subtyping is a relation on interface specifications. In most papers considering semantic subtyping, we find definitions similar to the following (cf. e.g. [LW94]):

**Definition 2.2 :** Semantic Subtype Relation
A partial ordering $\preceq$ on a set of interface specifications is called a *semantic subtype relation* iff the corresponding set of syntactic interfaces is well–formed and the following holds for all S,T with S $\preceq$ T:

1. The invariant of interface S implies the invariant of interface T. More precisely: Let $INV_{\mathrm{S}}(X,\$)$ be the invariant of S and $INV_{\mathrm{T}}(X,\$)$ be the invariant of T, then the following implication holds for all objects $XS$ of type S:

   $$INV_{\mathrm{S}}(XS,\$) \;\Rightarrow\; INV_{\mathrm{T}}(XS,\$)$$

2. If a method m is associated with S and T, S::m has to show the same behaviour as T::m, i.e. the precondition of T::m implies the precondition of S::m and the postcondition of S::m implies the postcondition of T::m.

□

Informally, the above definition means that (1.) objects of a subtype satisfy at least the properties of the objects of its supertypes and that (2.) a method of a subtype can be used in all contexts where the corresponding supertype method is allowed and execution of the subtype method establishes the postcondition of the supertype method. Although informally fairly clear, a closer look at such definitions turns up several vague aspects. We shortly discuss here two minor and one major aspect in order to make the above definition more precise and to motivate and prepare the formal analysis in the following chapters.

The first aspect concerns the requirement that preconditions of supertype methods have to imply corresponding preconditions in subtype methods. As above we assume that S and T are types, $\mathrm{S} \prec \mathrm{T}$, and m is a method associated with S and T. If a precondition of S::m requires the self–parameter to be of type S, which is trivially satisfied in all prestates of S::m, the corresponding precondition of T::m has to enforce as well that the self–parameter is of type S, which is in general not true for T::m. I.e. the above constraints are too strict (cf. the discussion of the subtype–rule in section 4.1.3).

The second aspect concerns the format in which specifications are given[20]. In section 2.2, we have discussed different equivalent formats in which method specification can be given. Even for equivalent specifications of the same method, the constraints on pre– and postconditions cannot be shown without a programming logic. E.g. the invariance property of method append can be specified by one of the following equivalent pre–post–pairs:

```
meth  append( n: INT ): LIST
   pre   alive(X,$)
   post  $^ ≡_X $

meth  append( n: INT ): LIST
   post  alive(X,$^) => $^ ≡_X $
```

The trivial precondition of the second specification does not imply the precondition of the first. This problem can e.g. be overcome by formulating the method specification constraints of definition 2.2 in terms of normalform specifications. In order to be

[20]This aspect was pointed out by Prof. Broy in a discussion on an earlier version of this thesis.

a semantic subtype, a normalform specification of S::m has to imply a normalform specification of T::m. An even better formulation is obtained if we require that free variables in normalforms that are not program variables (e.g. $X$ in the above example) have to be universally quantified in normalforms.

The major aspect is concerned with subtyping and abstraction. As an example to explain this aspect, we consider the type LIST with subtypes CLIST and PLIST and assume that an object of type LIST is either a CLIST–object or a PLIST–object. More precisely, LIST has only CLIST and PLIST as direct subtypes and LIST is an *abstract type* , i.e. there are no objects that belong only to LIST, but not to one of its subtypes.

As example, we consider the specification of method append for the three interfaces. For interfaces LIST and CLIST it is the same as long as we consider the abbreviated form:

```
meth  append( n: INT ): SAME
   post  result! = app( n!^, self!^ )
```

Thus the constraints from definition 2.2 seem to be trivially satisfied for this case. But in the expanded form[21], we are confronted with two different abstraction functions for lists in the pre– and postcondition where $aL$ and $aC$ denote the abstraction of objects of type LIST and CLIST respectively and $aI$ denotes the abstraction of INT–objects:

```
meth  append( n: INT ): LIST
   post  aL(result,$) = app( aI(n^), aL(self^,$^) )

meth  append( n: INT ): CLIST
   post  aC(result,$) = app( aI(n^), aC(self^,$^) )
```

In particular, function $aL$ has to abstract as well objects of type PLIST. To prove the constraint on method specifications according to definition 2.2, we have to relate $aL$ and $aC$. As CLIST extends LIST only by new methods, we can assume that the range of $aC$ is as well sort *List* and that they yield the same result as long as they are applied to objects of type CLIST. Using a function *typ* that yields the type of an object, this can be stated by:

$$typ(XL) \preceq \text{CLIST} \;\Rightarrow\; aL(XL,E) = aC(XL,E)$$

Based on the assumption that programs are type safe (i.e. $typ(\text{self\^{}}) \preceq \text{CLIST}$ and $typ(\text{result}) \preceq \text{CLIST}$ are true), we can prove the constraint.

Interface PLIST does not simply extend the functionality of interface LIST. It maintains additional information, namely a position in the represented list. A list with position can be modeled by a pair of lists, the first component of which corresponds to the part of the list before the position and the second to the part after the position.

[21]Cf. section 2.2.2 for expanding method annotations.

The sort of list pairs is *PList* with pairing function *plist* and selectors *front* for the first component and *back* for the second component, i.e.:

**data type**
$PList \;=\; plist(\,front : List, back : List\,)$
**end data type**

$$\begin{aligned} pltol \;:\; & PList \rightarrow List \\ & pltol(PL) = conc(front(PL), back(PL)) \end{aligned}$$

The function *pltol* maps lists with position to lists without positions by concatenating the front and back part. As we want to specify the behaviour of interface PLIST by the abstract data type of lists with positions, the abstraction function *aP* for PLIST has to yield values of sort *PList*. The method specifications for PLIST have to explain whether and how the list position is modified. For PLIST::append we assume that the position is set to the beginning of the resulting list:

```
meth  append( n: INT ): PLIST
   post  aP( result, $ )
         = plist( empt, app(aI(n^),pltol(aP(self^,$^))) )
```

In order to prove the constraints on method specifications from definition 2.2 for PLIST::append, we have to understand the relation between PLIST and LIST. As it is typical for the relation between a sub– and a supertype, PLIST is a specialization of LIST. The corresponding relation between the values of sort *PList* and the values of sort *List* is realized by the function *pltol* forgetting the position in the list. Using *pltol*, we can express the relation between the abstraction functions *aL* and *aP*:

$$typ(XL) \preceq \mathrm{PLIST} \;\Rightarrow\; aL(XL, E) = pltol(aP(XL, E))$$

Based on type safety and the definition of *pltol*, we can prove the constraint for method append. In general, functions like *pltol* that map values belonging to the subtype to values belonging to the supertype have to be part of an interface specification (for CLIST this mapping is the identity).

The discussion of the shortcomings of definition 2.2 should motivate the development of a formal framework that enables to give a precise meaning to interface specifications, that supports logically sound transformations on interface specifications, and that makes it possible to further analyze the relation between interface invariants and method specifications (cf. section 4.2.3).

# Chapter 3

# Formal Foundation of Program Specification and Verification

This chapter provides the formal background and the fundamental techniques for specifying and verifying imperative, class–based programs. A class–based programming language is essentially an object–oriented language without subtyping and inheritance. The chapter introduces and formally specifies **a**n **i**mperative **c**lass-based **k**ernel **l**anguage, called AICKL, with recursive classes and recursive methods. In particular, it presents a formal model of the object environment and an axiomatic semantics of AICKL. Central properties of object environments are investigated and program properties that hold for all AICKL programs are proved. Especially, it is shown that AICKL programs are type safe.

## 3.1 An Imperative Class–Based Kernel Language

Specification and verification of programs can only be successful, if the used programming language is well–defined. This section introduces AICKL, the kernel of a class–based imperative programming language. The central programming constructs of AICKL are recursive classes with recursive methods. Object–oriented extensions to AICKL are presented in chapter 4. The specification of AICKL is given in three parts. The first part presents the syntax of AICKL. The second part develops a formal data and state model for AICKL. The third part provides an axiomatic specification of the dynamic semantics of AICKL.

### 3.1.1 The Syntax of the Kernel Language

An AICKL program is a set of class declarations of the following form:

```
class  T  is
    attr  $a_1$        : $T_{a_1}$
       ...
    attr  $a_{ma(T)}$ : $T_{a_{ma(T)}}$
    METHOD_LIST
end
```

A class declaration consists of an attribute list and a method list. It provides an implementation for type T. To ease formulation, we often identify classes with the types they implement; in particular, we speak of class T when we mean the class implementing type T. The class above declares the attributes $\mathrm{a}_1$,..., $\mathrm{a}_{ma(\mathrm{T})}$. Attributes have a *domain type* and a *range type*. The domain type of attribute $\mathrm{a}_i$ $(1 \leq i \leq ma(\mathrm{T}))$ is T, its range type is $\mathrm{T}_{\mathrm{a}_i}$. The types INT and BOOL are predeclared in AICKL. All other types occurring in an AICKL program must be implemented by exactly one class. As convention, type identifiers are written with capital letters.
Method declarations have the following form:

```
meth  m ( $p_1$: $T_{p_1}$ , .. ,$p_q$: $T_{p_q}$ ): $T_m$ is
  $v_1$: $T_{v_1}$ ; .. ; $v_r$: $T_{v_r}$ ;
  STAT
end
```

Such a method declaration declares the method identifier m, the implicit parameter self (cf. introduction), the explicit parameters $\mathrm{p}_i$, and the local variables $\mathrm{v}_j$. The (possibly compound) statement STAT is called the body of m. The list of local variables has to contain a variable result of type $\mathrm{T}_m$. The value of result is returned on return from calls to m; this way, an extra return statement is dispensable. As identifier and type of variable result is clear from the context, its declaration may be omitted. If the result of a method is irrelevant, the result type may be omitted as well; then, the type of the enclosing class is assumed as result type. By the *variables (of a method)*, we mean both parameters and local variables. The variable identifiers of a method must be distinct. The attribute and method identifiers of a class must be distinct and different from the implicitly declared methods (see below).

In addition to the explicitly declared methods, a class implicitly provides methods for reading and writing attributes, the method equ for comparing objects, the identity method idt returning the given object, and the method new creating new objects (independent of the self parameter). In summary, a class declared as above has the following *class interface* :

```
class interface  T  is
   //  explicitly declared methods
   meth  m ( $p_1$: $T_{p_1}$ , .. ,$p_q$: $T_{p_q}$ ): $T_m$
     ...
   //  read and write methods for each attribute
   meth  read_$a_1$ (): $T_{a_1}$
   meth  write_$a_1$ ( p: $T_{a_1}$ )
     ...
   meth  read_$a_{ma(T)}$ (): $T_{a_{ma(T)}}$
   meth  write_$a_{ma(T)}$ ( p: $T_{a_{ma(T)}}$ )
   //  implicitly declared methods
   meth  equ( p: T ): BOOL
   meth  idt(): T
   meth  new(): T
end
```

For the predeclared types BOOL and INT, we assume an appropriate set of methods containing a method not for BOOL denoting negation, and methods add and less for INT denoting addition and the less–comparison. In addition, both have the methods equ and idt.

AICKL only supports atomic expressions. An expression occurring in a method m is either a constant or a local variable or parameter of method m. The type BOOL has the constants true and false; type INT provides integer literals for each of its elements; for each class T there is a constant void(T). As all expressions are atomic and variables are typed, the type of an expression is obvious.

AICKL provides a loop statement, a conditional statement, sequential statement composition, and a call statement. These statements have the following form:

```
while EXP do STAT end
if EXP then STAT1 else STAT2 end
STAT1 ; STAT2
v  :=  EXP0 . m ( EXP1, .. , EXPq )
```

The following context conditions have to be satisfied by statements: The expression in a loop or conditional statement has to be of type BOOL. In call statements, the interface of the type of $\mathrm{EXP}_0$ must contain a method with identifier m and the correct number of parameters. The types of actual parameter expressions must equal the types of m's formal parameters. The left hand side variable v has to be a local variable of the method enclosing the call statement; its type must equal the result type of m.

Using a kernel language with only four statements simplifies the axiomatic semantics. E.g. if we had distinguished between methods with and without results and between methods with and without implicit parameter[1] in the kernel language, we would have had to deal with four different method formats in the axiomatic semantics. To ease notation and to come closer to notational conventions, AICKL supports the following syntactical abbreviations:

- A call to a method without result (see e.g. method updfst in interface LIST; p. 12) is written as "$\mathrm{EXP}_0.\mathrm{m}(\mathrm{EXP}_1, \ldots, \mathrm{EXP}_z)$". It is considered as a shorthand for "vdummy := $\mathrm{EXP}_0.\mathrm{m}(\mathrm{EXP}_1, \ldots, \mathrm{EXP}_z)$" where vdummy is a local variable of the enclosing method that is otherwise not used. In the short form the declaration of vdummy may be omitted.

- Instead of "void(T).m(...)" one may write "T::m(...)" (cf. section 2.1).

- Instead of "v := EXP.idt()" we usually write "v := EXP".

- Instead of "v := void(T).new()" we usually write "v := new T".

- Reading and writing object attributes may be written in the usual syntax, i.e. instead of "v := EXP.read_att()" we usually write "v := EXP.att" and instead of "v.write_att( EXP )" we usually write "v.att := EXP".

[1]Class methods, cf. section 2.1.

We decided to express assignments by calls to the implicitly declared method idt. This leads to a programming logic with less rules, stresses the fundamental role of method calls, and demonstrates that even the basic assignment operation can be specified with the presented techniques. Beside this, we believe that it is a good decision for object–oriented programming languages. In realistic extensions of the kernel language, classes could be defined that support controlled or extended forms of assignment[2]. E.g. each time an object is assigned to a variable a hidden reference counter could be incremented. This way an object would know how many references to it possibly exist in a program state.

Whereas the absence of complex expressions simplifies the logic for AICKL, it is fairly inconvenient for programmers. But it is not a real restriction. Each program with complex expressions[3], i.e. expressions that are build up from constants, variables, and method calls, can be automatically translated into a program containing only atomic expressions. As an example, we consider the fragment from section 2.1 that swaps the first two elements of a list:

```
tmp :=  l.first() ;
l.updfst( l.rest().first() ) ;
l.rest(). updfst( tmp )
```

For each occurrence of a non–atomic expression we introduce a new variable:

```
tmp    :=  l.first() ;
lrest1 :=  l.rest()   ;
lrfst  :=  lrest1.first() ;
l.updfst( lrfst ) ;
lrest2 :=  l.rest()   ;
lrest2 . updfst( tmp ) ;
```

If the expression in a conditional statement is complex, it is executed before the condition. If the expression in a loop is complex, it is executed before the loop and at the end of the loop body. This is illustrated with method concat in the following example.

**A List Implementation in AICKL** To illustrate the syntax of AICKL, we provide an implementation for singly linked lists. The class is called CLIST, because it implements the interface CLIST given in chapter 2. With this example, we finish the syntactical definition of AICKL. The next section specifies the data and state model. In section 3.1.3, the dynamic semantics of AICKL is defined by program axioms.

[2]E.g. in C++ and BETA, it is possible to redefine the assignment operator; cf. the discussion in [MMPN93] on pages 69ff.

[3]We assume that expressions have a well–defined evaluation order.

```
class  CLIST  is
   attr head : INT
   attr tail : CLIST

   meth  empty  (): CLIST is  result := void(CLIST)             end
   meth  isempty(): BOOL  is  result := self.equ( void(CLIST) ) end
   meth  first  (): INT   is  result := self.head               end
   meth  rest   (): CLIST is  result := self.tail               end
   meth  append( e: INT ): CLIST  is
      lv : CLIST ;
      lv := new CLIST ;
      lv.head := e ;
      lv.tail := self ;
      result  := lv
   end
   meth  updfst( e: INT ) is  self.head := e  end
   meth  copy(): CLIST    is   ...            end  // body omitted
   meth  concat( l: CLIST ): CLIST is
      lv: CLIST; lv1: CLIST;  ev: INT; bv: BOOL;

      bv :=  self.isempty() ;
      if  bv  then
         result :=  l
      else
         lv  :=  self ;
         lv1 :=  self.tail ;
         bv  :=  lv1.isempty() ;
         bv  :=  bv.not() ;
         while  bv  do
            lv  :=  lv1 ;
            lv1 :=  lv1.tail ;
            bv  :=  lv1.isempty() ;
            bv  :=  bv.not()
         end ;
         lv.tail := l ;
         result  :=  self
      end
   end
end
```

Figure 3.1: Implementation of interface CLIST

### 3.1.2 Data and State Model of the Kernel Language

This section defines the data and state model for the kernel language AICKL. Beside providing the formal basis for the subsequent sections, it illustrates as well a technique that is applicable to a wide class of languages. It demonstrates what the data and state model of a language is and how it can be specified in an abstract way. Beside syntactical differences, it is essentially the data and state model in which imperative languages are different. E.g. the data and state model of PASCAL and C are very different, because the cast and arithmetic operations on pointers in C make it necessary to provide a fairly elaborated storage model which is not needed for PASCAL (cf. [Bic95]).

In order to enable formal reasoning about programs with complex types, the data and state model of the language has to be formalized and several aspects of this formalization has to be made accessible for program verification. Thus, the specification of the data and state model should be carried out in the same framework as the interface specifications of programs. In our case, this is the framework of many–sorted first–order specifications with recursive data types (cf. chapter 1). For the formalization of the data and state model in our framework, we have to answer in particular two questions:

1. Do different program types correspond to different sorts in the data specification?
2. How are the states of objects formalized?

Concerning the first question, we decided to use one sort for all objects; typing of objects is expressed by a function *typ* that yields for each object its type (see below). This design decision has the following advantages:

1. Typing properties can be expressed and proved within the programming logic.
2. The model works as well for weakly, ill, or untyped languages.
3. Certain predicates and functions have to be applicable to all objects (see 3.2). Thus, a sort containing all objects is desirable anyway.
4. The state model gets simpler.
5. Language extension to object–orientation, i.e. introduction of subtyping, becomes relatively straightforward.

The third and fifth argument deserve some explanation. They only hold in this strong form if we assume a many–sorted specification framework. In a specification framework supporting polymorphism with a powerful class concept[4] it should be possible to express the polymorphic functions defined on all objects. And if the specification framework supports subtyping as well, a correspondence between program types and

[4]The class concept is needed to guarantee that types corresponding to types of the programming language are equipped with a basic set of operations.

types of the underlying specification framework can be established even for object–oriented programs (cf. [Bre91] for relating programming language subtyping to subtyping in the specification language). But from a practical point of view it is important to not assume too much about the specification language. The techniques presented in this thesis are only applicable to realistic size programs if they are supported by a powerful computer–based proof environment. To exploit the technology provided by the best existing provers, we have to be able to build on their specification languages. And many of them do not support subtyping.

Concerning the second question, we use an abstract data type ObjEnv. Values of sort *ObjEnv* record for each object its state and whether the object is alive or not. Object states are modeled via *object locations*: For each attribute of its class, an object has a location. Object locations correspond to what is called instance variables in Smalltalk and Java, slots in Self, data members in C++, and fields in Eiffel.

The definition of typed objects and *object environments* depends on the set of types and attributes and their relation. The simplest approach is to assume that a program is given and to define the objects and object environments for this given program. The disadvantage of this approach turns up when programs are extended. Extending programs usually means that new types and attributes are declared; i.e. the set of objects is enlarged. Thus, formulas quantifying over all objects that could be proved to be valid in the original program may become invalid in the extended program (cf. chapter 4). To avoid this disadvantage, we consider a program as part of all its extensions. I.e. we assume an infinite sort *TypId* of type identifiers and an infinite sort *AttId* of attribute identifiers where attributes of different classes with the same name are considered to be different (e.g. by prefixing them with the type identifier; see below). In this approach, a program determines the relation between its types and attributes, but leaves the relation between types and attributes not occurring in the program un(der)specified. Extending a program in this approach means to refine the type–attribute–relation. In addition to *TypId* and *AttId*, we assume an infinite sort *ObjId* of object identifiers. Based on these sorts, the data model is specified as follows (cf. section 1.3 for an introduction to the data type construct):

$$
\begin{array}{l}
\textbf{data type} \\
\quad \begin{array}{lcl}
\mathit{Type} & = & \mathit{BOOL} \\
 & | & \mathit{INT} \\
 & | & \mathit{ct}(\ \mathit{TypId}\ ) \\
\mathit{Object} & = & \mathit{bool}(\mathit{Boolean}) \\
 & | & \mathit{into}(\mathit{Int}) \\
 & | & \mathit{void}(\ \mathit{TypId}\ ) \\
 & | & \mathit{object}(\ \mathit{TypId},\ \mathit{ObjId}\ )
\end{array} \\
\textbf{end data type}
\end{array}
$$

For the BOOL–objects *bool*(FALSE) and *bool*(TRUE) we use the abbreviations *false* and *true*. In program texts, INT–objects are written by their usual digital representation. In formulas, we have to distinguish between INT–objects and values of sorts *Int* and *Integer*: Values of sort *Integer* are written as usual, values of sort *Int* are denoted by using the function *intc* (cf. section 1.3), and INT–objects are denoted by using the constructor *into*. As abbreviation, we introduce a direct mapping from integers to INT–objects:

$$\begin{array}{l} int : Integer \;\rightarrow\; Object \\ \quad int(I) = into(intc(I)) \end{array}$$

The standard abstraction functions on BOOL– and INT–objects are defined as follows:

$$\begin{array}{l} aB : Object \;\rightarrow\; Boolean \\ \quad aB(bool(B)) = B \\ aI : Object \;\rightarrow\; Integer \\ \quad minint \leq I \leq maxint \;\Rightarrow\; aI(int(I)) = I \end{array}$$

As can be seen from the definition of data type *Object*, an object of a declared type is either the void object or a typed object identifier. As mentioned above, the state of an object is modeled via so–called *locations*. (The term "location" is taken from the literature about programming language semantics; cf. e.g. [Win93].) A location can be considered as an anonymous variable, i.e. a variable that can only be referenced through the object it belongs to. An object possesses a location for each attribute of its class:

**data type**
$Location \;=\; location(\, AttId,\; ObjId\,)$
**end data type**

The relations between objects, types, and attributes and their basic properties are expressed by the following functions: *typ* yields the type of an object; *isvoid* asserts that an object is a void object; *dtyp* and *rtyp* yield the domain and range type of an attribute; *ltyp* yields the type of a location; *obj* yields the object a location belongs to; given an object and an attribute, *loc* yields the corresponding location; *static* asserts that an object is not subject to object creation. Except *dtyp* and *rtyp*, all functions are program independent. They are specified as follows:

$$\begin{array}{lcl} \multicolumn{3}{l}{typ : Object \rightarrow Type} \\ \quad typ(bool(B)) & = & BOOL \\ \quad typ(into(I)) & = & INT \\ \quad typ(void(T)) & = & ct(T) \\ \quad typ(object(T, OID)) & = & ct(T) \end{array}$$

$$\begin{array}{l} isvoid : Object \rightarrow Boolean \\ \quad \neg isvoid(bool(B)) \\ \quad \neg isvoid(into(I)) \\ \quad \neg isvoid(object(T, OID)) \\ \quad isvoid(void(T)) \end{array}$$

$$\begin{array}{l} ltyp : Location \;\rightarrow\; Type \\ \quad ltyp(location(A, OID)) \;=\; rtyp(A) \end{array}$$

$$\begin{array}{l} obj : Location \ \rightarrow \ Object \\ \quad obj(location(A, OID)) \ = \ object(dtyp(A), OID) \end{array}$$

$$\begin{array}{l} loc : Object \times AttId \ \rightarrow \ Location \\ \quad loc(object(dtyp(A), OID), A) = location(A, OID) \end{array}$$

$$\begin{array}{l} init : Type \ \rightarrow \ Object \\ \quad \begin{array}{lcl} init(BOOL) & = & false \\ init(INT) & = & int(0) \\ init(ct(T)) & = & void(T) \end{array} \end{array}$$

$$\begin{array}{l} static : Object \ \rightarrow \ Boolean \\ \quad static(X) \ \Leftrightarrow \ typ(X) = BOOL \vee typ(X) = INT \vee isvoid(X) \end{array}$$

$$\begin{array}{l} dtyp : AttId \ \rightarrow \ TypId \\ rtyp : AttId \ \rightarrow \ Type \end{array}$$

The domain and range types of attributes are program dependent, but do not change if a program is extended. We assume that *dtyp* and *rtyp* are specified for all attributes of a given program by enumeration.

The object environment is modeled by an abstract data type with main sort *ObjEnv* and the following operations: $E\langle L := X\rangle$ denotes updating the object environment $E$ at location $L$ with object $X$. $E(L)$ denotes reading location $L$ in environment $E$; $E(L)$ is called the *object held by L in E*. $new(E, T)$ returns a new object of type $T$ in environment $E$. $E\langle T\rangle$ denotes the environment after allocating a new object of type $T$ in $E$. $alive(X, E)$ yields true iff object $X$ is alive in $E$:

$$\begin{array}{ll} \_\langle\_ := \_\rangle & : ObjEnv \times Location \times Object \ \rightarrow \ ObjEnv \\ \_\langle\_\rangle & : ObjEnv \times TypId \ \rightarrow \ ObjEnv \\ \_(\_) & : ObjEnv \times Location \ \rightarrow \ Object \\ alive & : Object \times ObjEnv \ \rightarrow \ Boolean \\ new & : ObjEnv \times TypId \ \rightarrow \ Object \end{array}$$

In the following, we present and explain the axiomatization of these functions. Location update $\_\langle\_ := \_\rangle$ and object allocation $E\langle T\rangle$ construct new environments from given ones, location read and liveness test allow to observe environments. We first consider the properties of environments that can be observed by location reads, then the liveness properties, and finally the properties of the *new*–operation and an extensionality axiom.

Axiom env1 states that updating one location does not affect the objects held by other locations. Axiom env2 states that reading a location updated by an object $X$ yields $X$, if the object of the location and $X$ are alive. We restrict this property to living objects in order to guarantee that locations never hold non–living objects and that locations of non–living objects are initialized as described by axiom env3. Axiom env4 states that updates by non–living objects do not modify the environment. From a formal point of view this axiom is not needed, because in AICKL only living objects can be referenced so that updates with non–living objects cannot occur in programs.

It is introduced to simplify notions, lemmas, and proofs later on[5]. Axiom env5 states that allocation does not affect the objects held by locations:

$$\begin{aligned}
&\text{env1}: && \mathit{L1} \neq \mathit{L2} \;\Rightarrow\; E\langle \mathit{L1} := X\rangle(\mathit{L2}) = E(\mathit{L2})\\
&\text{env2}: && \mathit{alive}(\mathit{obj}(L), E) \wedge \mathit{alive}(X, E) \;\Rightarrow\; E\langle L := X\rangle(L) = X\\
&\text{env3}: && \neg\mathit{alive}(\mathit{obj}(L), E) \;\Rightarrow\; E(L) = \mathit{init}(\mathit{ltyp}(L))\\
&\text{env4}: && \neg\mathit{alive}(X, E) \;\Rightarrow\; E\langle L := X\rangle = E\\
&\text{env5}: && E\langle T\rangle(L) = E(L)
\end{aligned}$$

Axiom env6 states that location updates have no influence on liveness of objects. Axiom env7 specifies that an object is alive after allocation iff it is alive before allocation or it is the newly allocated object. To make proofs simpler, axiom env8 enforces that objects held by locations are alive. Finally, static objects, i.e. objects that are not subject to creation, are considered to be alive:

$$\begin{aligned}
&\text{env6}: && \mathit{alive}(X, E\langle L := Y\rangle) \;\Leftrightarrow\; \mathit{alive}(X, E)\\
&\text{env7}: && \mathit{alive}(X, E\langle T\rangle) \;\Leftrightarrow\; \mathit{alive}(X, E) \vee X = \mathit{new}(E, T)\\
&\text{env8}: && \mathit{alive}(E(L), E)\\
&\text{env9}: && \mathit{static}(X) \;\Rightarrow\; \mathit{alive}(X, E)
\end{aligned}$$

The following three axioms specify the properties of the new–operation. A newly created object is not alive in the environment in which it is created (env10). A newly created object has the correct type (env11). The creation of an object of type T yields the same result in two environments if and only if liveness for T–objects is equivalent in these environments:

$$\begin{aligned}
&\text{env10}: && \neg\mathit{alive}(\mathit{new}(E, T), E)\\
&\text{env11}: && \mathit{typ}(\mathit{new}(E, T)) = \mathit{ct}(T)\\
&\text{env12}: && \mathit{new}(\mathit{E1}, T) = \mathit{new}(\mathit{E2}, T)\\
& && \quad\Leftrightarrow\; \forall X : \mathit{typ}(X) = \mathit{ct}(T) \;\Rightarrow\; (\mathit{alive}(X, \mathit{E1}) \;\Leftrightarrow\; \mathit{alive}(X, \mathit{E2}))
\end{aligned}$$

Finally, we guarantee that two environments are equal if we cannot distinguish them by the observer functions:

$$\begin{aligned}
&\text{env13}: && (\forall X : \mathit{alive}(X, \mathit{E1}) \;\Leftrightarrow\; \mathit{alive}(X, \mathit{E2})) \wedge (\forall L : \mathit{E1}(L) = \mathit{E2}(L))\\
& && \quad\Rightarrow\; \mathit{E1} = \mathit{E2}
\end{aligned}$$

A model for these axioms is presented in appendix A.1. As an example how these axioms can be used we prove the following lemma:

**Lemma 3.1 :** Commutativity of Object Creation and Location Update
The creation of an object and the update of a location can be exchanged if the updated location is not a location of the newly created object and if the update does not concern the new object:

$$\mathit{obj}(L) \neq \mathit{new}(E, T) \wedge X \neq \mathit{new}(E, T) \;\Rightarrow\; E\langle T\rangle\langle L := X\rangle = E\langle L := X\rangle\langle T\rangle$$

□

[5] E.g. in lemma 3.1 the free variable $X$ would otherwise have to be restricted to living objects.

**Proof**
To apply axiom env13, we have to show that (i) objects alive in $E\langle T\rangle\langle L := X\rangle$ are alive in $E\langle L := X\rangle\langle T\rangle$ and vice versa and that (ii) locations hold the same objects in these environments:

(i) Let $Y$ be an arbitrary object:

$$\begin{array}{cl}
 & alive(Y, E\langle T\rangle\langle L := X\rangle) \\
\Leftrightarrow & \quad [\![ \text{env6} ]\!] \\
 & alive(Y, E\langle T\rangle) \\
\Leftrightarrow & \quad [\![ \text{env7} ]\!] \\
 & alive(Y, E) \vee Y = new(E, T) \\
\Leftrightarrow & \quad [\![ \text{env6} ]\!] \\
 & alive(Y, E\langle L := X\rangle) \vee Y = new(E, T) \\
\Leftrightarrow & \quad [\![ \text{env12 with env6} ]\!] \\
 & alive(Y, E\langle L := X\rangle) \vee Y = new(E\langle L := X\rangle, T) \\
\Leftrightarrow & \quad [\![ \text{env7} ]\!] \\
 & alive(Y, E\langle T\rangle\langle L := X\rangle)
\end{array}$$

(ii) Let $K$ be an arbitrary location. If $K \neq L$, the axioms env1 and env5 allow to derive:

$$E\langle T\rangle\langle L := X\rangle(K) = E\langle T\rangle(K) = E(K) = E\langle L := X\rangle(K) = E\langle L := X\rangle\langle T\rangle(K)$$

For the case $K = L$ we need the premise $obj(L) \neq new(E, T) \wedge X \neq new(E, T)$ that allows to derive with env7:

$$alive(obj(L), E\langle T\rangle) \quad \Leftrightarrow \quad alive(obj(L), E) \qquad (3.1)$$
$$alive(X, E\langle T\rangle) \quad \Leftrightarrow \quad alive(X, E) \qquad (3.2)$$

We consider three case:

a) Assuming $alive(obj(L), E)$ and $alive(X, E)$, equivalences 3.1 and 3.2 yield $alive(obj(L), E\langle T\rangle)$ and $alive(X, E\langle T\rangle)$. By applying two times axiom env2 and once env5 results in the following equation chain:

$$E\langle T\rangle\langle L := X\rangle(L) = X = E\langle L := X\rangle(L) = E\langle L := X\rangle\langle T\rangle(L)$$

b) Assuming $\neg alive(obj(L), E)$, equivalence 3.1 yields $\neg alive(obj(L), E\langle T\rangle)$. Applying two times env3 (together with env6) and then env5, we get:

$$E\langle T\rangle\langle L := X\rangle(L) = init(ltyp(L)) = E\langle L := X\rangle(L) = E\langle L := X\rangle\langle T\rangle(L)$$

c) Assuming $\neg alive(X, E)$, equivalence 3.2 yields $\neg alive(X, E\langle T\rangle)$; applying env4 and env5 two times yields:

$$\begin{aligned} E\langle T\rangle\langle L := X\rangle(L) &= E\langle T\rangle(L) = E(L) \\ &= E\langle L := X\rangle(L) = E\langle L := X\rangle\langle T\rangle(L) \end{aligned}$$

**end of proof**

**Program Specific Aspects of Signatures and States** To specify properties of a given program $\mathcal{P}$, we have to refer to variables, attributes, and types in formulas. This is enabled by introducing constant symbols for all these entities. To define this more precisely, let $\mathcal{P}$ be an AICKL program and let $\Sigma_{general}$ denote a signature that includes the signature of the data and state model as introduced above. We assume that the constant symbols of $\Sigma_{general}$ and the identifiers occurring in $\mathcal{P}$ are distinct. For each type T declared in $\mathcal{P}$, we add a constant symbol T of sort *TypId* to $\Sigma_{general}$, and for each attribute att declared in class T, we add a constant symbol denoted by T::att. The resulting signature is $\Sigma(\mathcal{P})$ or simply $\Sigma$ if $\mathcal{P}$ is clear from the context.

To refer to the current object environment the constant symbol \$ of sort *ObjEnv* is used, and $\Gamma$ denotes $\Sigma \cup \{\$\}$. The current object environment \$ can be considered as a global variable. In programs, \$ can only be accessed and modified through the read and write operations of attributes and the new operation. Specifications may directly refer to \$ (cf. chapter 2). Furthermore, we introduce three signatures for each method m occurring in $\mathcal{P}$: 1. The extension of $\Gamma$ by constant symbols of sort *Object* for each parameter (in particular, self) is denoted by $\Gamma_{pre(m)}$. 2. The extension of $\Gamma$ by constant symbols of sort *Object* for each parameter and local variable is denoted by $\Gamma_{body(m)}$. 3. The extension of $\Gamma$ by the constant symbol result of sort *Object* is denoted by $\Gamma_{post}$ (cf. section 2.2.2).

If we talk informally about *execution points*, we mean the positions before and after statements and methods. A *state* is characterized by the current execution point and by the interpretation of the constant symbols representing variables. In particular, a *prestate* is a state before the execution of a method's body and provides interpretations for the parameters and the object environment \$; a *poststate* is a state after the execution of a method and provides interpretations for the local variable result and for \$.

To illustrate the introduced notions, we consider the following statement sequence from method append of class CLIST given in section 3.1.1:

```
lv := new CLIST ;
lv.head := e ;
lv.tail := self
```

If $E$ is the value of the object environment before the execution of the statements, the value of the object environment variable \$ after the execution of the statements is given by[6]:

$$\$ = E\langle ct(\text{CLIST})\rangle\langle loc(\text{lv}, \text{CLIST::head}) := \text{e}\rangle\langle loc(\text{lv}, \text{CLIST::tail}) := \text{self}\rangle$$

For convenience, we use the following abbreviations in the rest of this thesis:

1. If the class of an attribute is clear from the context, the prefixed class identifier can be dropped, i.e. one may write "head" instead of "CLIST::head".

2. In $ct(T)$, the constructor $ct$ may be dropped if it is clear from the context whether $T$ is of sort *TypId* or of sort *Type*.

[6]How this can be formally derived is the topic of the following sections.

3. For the binary function *loc*, we support the infix syntax $\_.\_$, i.e. for "$loc(\mathrm{lv}, \mathrm{head})$" one may write "lv.head".

4. In sequences of location updates and object creations, the semicolon can be used as separator instead of the closing and opening angle brackets.

Applying all these abbreviations makes the above example more readable:

$\$ = E\langle \mathrm{CLIST};\ \mathrm{lv.head} := \mathrm{e};\ \mathrm{lv.tail} := \mathrm{self}\rangle$

### 3.1.3 Axiomatic Semantics of the Kernel Language

This section specifies the dynamic behaviour of AICKL programs by program axioms and rules. It defines the precise form of axioms and rules as well as their meaning and briefly discusses the axioms and rules for AICKL.

A *program component* is a statement occurrence or a method declaration occurrence within a given program. I.e. for program components we always assume that the program context in which they occur is implicitly given. In particular, we can refer to the method enclosing a statement. If the class T in which a method m is declared is not clear from the context, we prefix the method identifier with the type identifier, i.e. write T::m. A *Hoare triple* or simply *triple* has the form { **P** } COMP { **Q** } where COMP is a program component and **P** and **Q** are first–order formulas, called *pre*– and *postconditions* respectively. If the component in a triple **A** is a method, we call **A** a *method annotation*; otherwise **A** is called a *statement annotation* . Pre– and postconditions of statement annotations are formulas over $\Gamma_{body(m)}$ where m is the enclosing method; pre– and postconditions in annotations of method m are $\Gamma_{pre(m)}$–formulas and $\Gamma_{post}$–formulas respectively. As abbreviation, the prestate–operator may be used in postconditions (cf. section 2.2.1 for its meaning and section 2.2.2 for its elimination).

A triple { **P** } COMP { **Q** } specifies the following *refined* partial correctness property: If **P** holds in a state before executing COMP, then execution of COMP either

1. terminates and **Q** holds in the state after execution or

2. aborts because of errors or actions that are beyond the semantics of the programming language (e.g. memory allocation problems, stack overflow, external interrupts from the execution environment) or

3. runs forever.

In particular, execution of COMP does not abort because of (illegal) dereferencing of void objects or arithmetic exceptions. Thus, this refined partial correctness logic can be used to prove the absence of illegal dereferencing and arithmetic exceptions that are a frequent source of program errors. The logic lies inbetween the classical notion of partial correctness and total correctness. From a theoretical point of view, it has the disadvantages that it is more difficult to provide a semantics for this logic than for the other two candidates and that it is more complex than partial correctness (e.g. the triple $\{\ \mathbf{P}\ \}$ COMP $\{$ TRUE $\}$ is in general not valid). From a practical point of view, it is a good compromise: A classical partial correctness logic is not capable to detect the most frequent errors in sequential programs (arithmetic overflow, array bound violation, illegal dereferencing). A total correctness logic in the theoretical sense is not realizable, because nowadays programming languages do not control memory usage and interrupts from the environment; thus, a program that is proved total correct may abort because of aspects listed in point 2 above. Certainly, a logic that eliminates point 3 of the list above would be more valuable[7]. However, for the purposes of this thesis, a concentration to the object–oriented aspects seemed appropriate to us.

A *sequent* has the form $\mathcal{A} \vdash \mathbf{A}$ where $\mathcal{A}$ is a set of method annotations and $\mathbf{A}$ is a triple. Triples in $\mathcal{A}$ are called *assumptions* of the sequent and $\mathbf{A}$ is called the *consequent* of the sequent. A sequent expresses the fact that we can prove a triple based on some assumptions about methods. Sequents are necessary to handle recursive procedures and allow to formulate and prove properties of programs that use not yet implemented methods. Triples can be considered as sequents without assumptions.

The axiomatic semantics of AICKL consists of axioms for the implicitly and predeclared methods, axioms for the read and write methods of attributes, and rules explaining statement and method behaviour. The axiomatic semantics is designed so that it reflects the intuitive meaning of program components. The use of axioms and rules for program verification is discussed in section 3.3.

**Semantics of Basic Methods** For better readability, the axioms are presented by using the prestate–operator (a version without prestate–operator is contained in appendix A.2). They simply formulate the method semantics in terms of the abstract data and environment model:

equ–axiom:

$$\vdash \{\ \mathrm{TRUE}\ \}\ \ \mathrm{meth\ T::equ(p)}\ \ \{\ \mathrm{result} = \mathit{bool}(\mathrm{self}\hat{} = \mathrm{p}\hat{}\,)\ \wedge\ \$ = \$\hat{}\ \}$$

idt–axiom:

$$\vdash \{\ \mathrm{TRUE}\ \}\ \ \mathrm{meth\ T::idt()}\ \ \{\ \mathrm{result} = \mathrm{self}\hat{}\ \wedge\ \$ = \$\hat{}\ \}$$

new–axiom:

$$\vdash \{\ \mathrm{TRUE}\ \}\ \ \mathrm{meth\ T::new()}\ \ \{\ \mathrm{result} = \mathit{new}(\$\hat{}, \mathrm{T}) \wedge \$ = \$\hat{}\langle \mathrm{T}\rangle\ \}$$

Whereas in the above axioms the precondition is trivial, the precondition for method add and for the read and write methods of attributes formulate requirements that must hold in the prestate. Thus the axiomatic semantics of AICKL is a loose semantics, as it does not specify what happens in case these requirements are not satisfied. In

[7]The theory to extend the presented logic to such a logic is the same as to extend a classical partial correctness logic to a total correctness logic.

particular, we do not know if a program aborts in case of arithmetic overflow or dereferencing of a void–object or if it simply continues with unspecified results.

add–axiom:

$$\vdash \{\ \mathit{minint} \leq aI(\text{self}) + aI(\text{p}) \leq \mathit{maxint}\ \}$$
$$\text{meth INT::add(p)} \quad \{\ \text{result} = \mathit{int}(aI(\text{self}\hat{}) + aI(\text{p}\hat{})) \wedge \$ = \$\hat{}\ \}$$

less–axiom:

$$\vdash \{\ \text{TRUE}\ \} \quad \text{meth INT::less(p)} \quad \{\ \text{result} = \mathit{bool}(aI(\text{self}\hat{}) < aI(\text{p}\hat{})) \wedge \$ = \$\hat{}\ \}$$

not–axiom:

$$\vdash \{\ \text{TRUE}\ \} \quad \text{meth BOOL::not()} \quad \{\ \text{result} = \mathit{bool}(\neg aB(\text{self}\hat{})) \wedge \$ = \$\hat{}\ \}$$

In the axioms for the read and write methods, we use the abbreviation for attributes introduced at the end of section 3.1.2; in particular, we write self.att for $\mathit{loc}(\text{self}, \text{T::att})$. Both axioms require that the self–parameter is non–void and of the correct type:

read–att–axiom:

$$\vdash \{\ \neg \mathit{isvoid}(\text{self}) \wedge \mathit{typ}(\text{self}) = \text{T}\ \}$$
$$\text{meth T::read\_att()} \quad \{\ \text{result} = \$\hat{}(\text{self}\hat{}.\text{att}) \wedge \$ = \$\hat{}\ \}$$

write–att–axiom:

$$\vdash \{\ \neg \mathit{isvoid}(\text{self}) \wedge \mathit{typ}(\text{self}) = \text{T}\ \}$$
$$\text{meth T::write\_att(p)} \quad \{\ \$ = \$\hat{}\langle \text{self}\hat{}.\text{att} := \text{p}\hat{} \rangle\ \}$$

**Statement Semantics** The semantics of statements and methods is specified by a set of rules. A *rule* consists of a number of antecedents and a sequent as *succedent*. The *antecedents* are first–order formulas or sequents. Rules may contain metavariables for formulas and assumptions (the application of rules in verification is explained in section 3.3). A rule allows to prove the succedent from the antecedents. The semantical interpretation of rules is that each operational semantics (and each implementation) of the specified language has to guarantee the correctness of the rule w.r.t. the operational semantics (and implementation respectively).

The rules for the loop, conditional, and sequential statement are canonical except that we have to map the BOOL–objects occurring in the conditions to boolean values:

while-rule:

$$\frac{\mathcal{A} \vdash \{\ aB(\text{EXP}) \wedge \mathbf{P}\ \}\ \text{STAT}\ \{\ \mathbf{P}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{while EXP do STAT end}\ \{\ \neg aB(\text{EXP}) \wedge \mathbf{P}\ \}}$$

if–rule:

$$\frac{\begin{array}{l}\mathcal{A} \vdash \{\ aB(\text{EXP}) \wedge \mathbf{P}\ \}\ \text{STAT1}\ \{\ \mathbf{Q}\ \} \\ \mathcal{A} \vdash \{\ \neg aB(\text{EXP}) \wedge \mathbf{P}\ \}\ \text{STAT2}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{if EXP then STAT1 else STAT2 end}\ \{\ \mathbf{Q}\ \}}$$

seq-rule:

$$\frac{\begin{array}{l}\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{STAT1}\ \{\ \mathbf{Q}\ \} \\ \mathcal{A} \vdash \{\ \mathbf{Q}\ \}\ \text{STAT2}\ \{\ \mathbf{R}\ \}\end{array}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \text{STAT1 ; STAT2}\ \ \{\ \mathbf{R}\ \}}$$

As expected, the semantics of method calls is obtained by substituting the formal by the actual parameters:

call–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \text{meth T::m}(\text{p}_1, \ldots, \text{p}_z)\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[\text{E}_0/\text{self}, \text{E}_1/\text{p}_1, \ldots, \text{E}_z/\text{p}_z]\ \}\ \text{v} := \text{E}_0\ .\ \text{m}(\text{E}_1, \ldots, \text{E}_z)\ \{\ \mathbf{Q}[\text{v}/\text{result}]\ \}}$$

The call–rule only adapts the method specification to the actual parameters of the call; in particular, it specifies the assignment to left hand side variable v. The fact that variables different from v are not modified is expressed by the following rule. It allows to substitute these variables for free logical variables:

var–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{v} := \text{E}_0\ .\ \text{m}(\text{E}_1, \ldots, \text{E}_z)\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[\text{w}/Z]\ \}\ \text{v} := \text{E}_0\ .\ \text{m}(\text{E}_1, \ldots, \text{E}_z)\ \{\ \mathbf{Q}[\text{w}/Z]\ \}}$$

where w is a local variable or parameter of the method enclosing the call different from v.

The following, last rule explains method annotations. Essentially, an annotation holds for a method m if it holds for the body of m. This basic rule is strengthened in two aspects. In order to handle recursion, the method annotation may be assumed for the proof of the body. Informally, this is because in any terminating execution, the last incarnation does not contain a recursive call of the method. The termination case can be considered as a base case in an inductive proof. For this case the assumption is not needed, but established as induction hypothesis. For a non–terminating call this induction hypothesis may be assumed. The second aspect concerns the values of the local variables in the prestate. We assume here that they are initialized to the initial object of their type. This language design decision might cause inefficiencies in implementations, but a decision of that kind is mandatory to prove type safety and to show that only living objects can be referenced (see section 3.4).

rec–rule:

$$\frac{\begin{array}{l}\{\ \mathbf{P}\ \}\ \ \text{meth T::m}(\text{p}_1,\dots,\text{p}_z)\ \{\ \mathbf{Q}\ \}\ ,\mathcal{A}\\ \quad\vdash\ \{\ \mathbf{P}\wedge\bigwedge_i \text{v}_i = \mathit{init}(\text{TV}_i)\ \}\ \text{BODY(T::m)}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A}\vdash\ \{\ \mathbf{P}\ \}\ \ \text{meth T::m}(\text{p}_1,\dots,\text{p}_z)\ \{\ \mathbf{Q}\ \}}$$

where $\text{v}_i$ are the local variables of method T::m with type $\text{TV}_i$ and BODY(T::m) denotes the body of T::m.

This ends the definition of the AICKL language. The following sections investigate properties of environments and present a programming logic that allows to apply the above rules for program verification.

## 3.2 Properties of Object Environments

For the specification of relations between object environments in the prestate and poststate of methods, this section introduces predicates expressing properties of object environments. Because these predicates are relevant for all programs of a language, the basic properties of these predicates are investigated and presented in some detail. In particular, the operations on environments used in chapter 2 are defined, i.e. disjointness of objects in an environment and $X$–equivalence of two environments.

An important aspect of proving programs correct is to show that under certain conditions a property remains invariant if a location is updated or an object is created. To analyze these conditions, we assume that the property is given by a predicate $p(X,E)$ where $X$ is of sort *Object* and $E$ of sort *ObjEnv*. The invariants explained in section 2.3.2 are examples of such predicates. The fact that $p(X,E)$ is invariant under condition $\mathbf{Q}$ during an update to location $L$ by object $Z$ can be formulated by the following implication:

$$p(X,E)\wedge\mathbf{Q}\ \Rightarrow\ p(X,E\langle L:=Z\rangle)$$

In practice, $p(X,E)$ depends only on objects held by those locations in $E$ that are reachable from $X$ in $E$. Thus, a typical candidate for condition $\mathbf{Q}$ is non–reachability of the updated location:

$$p(X,E)\wedge\neg\mathit{reach}(X,L,E)\ \Rightarrow\ p(X,E\langle L:=Z\rangle)$$

Reachability of locations is such an important aspect in verification of programs with complex data types that some frameworks build it into their state model (cf. [Möl93]). In our approach the fact that the reachability predicate has to be applicable to all objects and all locations influenced the decision to use one sort *Object* for objects of all types (cf. section 3.1.2).

The definition of reachability is recursive: A location $L$ is called *reachable* from $X$ if $L$ is a location of $X$ or if there exists a location $K$ of $X$ such that $L$ is reachable from the object held by $K$ in $E$. There are essentially two ways to formalize such recursive definitions within first–order syntax so that inductive proofs are possible.

One possibility is to use a fixpoint operator and allow computational induction. The other possibility is to extend the signature of the recursively defined predicates by an additional parameter the data type of which is equipped with a well–founded ordering. For the purposes of this thesis, the latter approach is sufficient and technically simpler. In all cases, we can use an additional parameter of data type Nat. Applying this technique, we define a predicate $reach_n$ expressing that an object reaches a location in $N$ steps:

$$\begin{array}{ll} reach_n : Object \times Location \times Nat \times ObjEnv \rightarrow Boolean & \\ \quad reach_n(X, L, 0, E) & \Leftrightarrow X = obj(L) \\ \quad reach_n(X, L, N+1, E) & \Leftrightarrow \exists K : obj(K) = X \wedge reach_n(E(K), L, N, E) \end{array}$$

$$\begin{array}{ll} reach : Object \times Location \times ObjEnv \rightarrow Boolean & \\ \quad reach(X, L, E) & \Leftrightarrow \exists N : reach_n(X, L, N, E) \end{array}$$

The properties of *reach* that are needed within this thesis are summarized by the following lemma:

**Lemma 3.2 :** Properties of Predicate *reach*

i) $\neg reach(X, L, E) \Rightarrow (reach(X, K, E) \Leftrightarrow reach(X, K, E\langle L := Y\rangle))$

ii) $\neg reach(X, L, E) \Rightarrow \neg reach(X, L, E\langle L := Y\rangle)$

iii) $\neg reach(Y, K, E) \wedge \neg reach(X, K, E) \Rightarrow \neg reach(X, K, E\langle L := Y\rangle)$

iv) $\neg reach(obj(dtyp(A), OID), L, E) \Rightarrow loc(A, OID) \neq L$

v) $reach(X, L, E) \Leftrightarrow reach(X, L, E\langle T\rangle)$

vi) $static(X) \Rightarrow \neg reach(X, L, E)$

vii) $\neg alive(X, E) \wedge reach(X, L, E) \Rightarrow X = obj(L)$

viii) $alive(X, E) \wedge reach(X, L, E) \Rightarrow alive(obj(L), E)$

ix) $(\forall L : reach(X, L, E) \Rightarrow E(L) = E'(L)) \Rightarrow \forall L : reach(X, L, E) \Leftrightarrow reach(X, L, E')$

□

**Proof**

The properties given in lemma 3.2 are proven by induction on the length of the reach chain. We demonstrate it here for the first property showing:

$$\neg reach(X, L, E) \Rightarrow (reach_n(X, K, N, E) \Leftrightarrow reach_n(X, K, N, E\langle L := Y\rangle))$$

by induction on N:

**Induction Base:** The definition of $reach_n$ yields directly:

$$reach_n(X, K, 0, E) \Leftrightarrow reach_n(X, K, 0, E\langle L := Y\rangle)$$

**Induction Step:** For the following derivation $\neg reach(X, L, E)$ is assumed:

$$\begin{array}{ll}
 & reach_n(X, K, N+1, E) \\
\Leftrightarrow & \quad [\![ \text{ definition of } reach_n ]\!] \\
 & \exists K : obj(K) = X \wedge reach_n(E(K), L, N, E) \\
\Leftrightarrow & \quad [\![ \text{ induction hypothesis } ]\!] \\
 & \exists K : obj(K) = X \wedge reach_n(E(K), L, N, E\langle L := Y\rangle) \\
\Leftrightarrow & \quad [\![ \neg reach(X, L, E) \wedge obj(K) = X \Rightarrow K \neq L \text{ and env1 } ]\!] \\
 & \exists K : obj(K) = X \wedge reach_n(E\langle L := Y\rangle(K), L, N, E\langle L := Y\rangle) \\
\Leftrightarrow & \quad [\![ \text{ definition of } reach_n ]\!] \\
 & reach_n(X, K, N+1, E\langle L := Y\rangle)
\end{array}$$

**end of proof**

Based on *reach* we define what it means that an object reaches another object and what it means that two objects are disjoint in an object environment. Both predicates are often more convenient to use than *reach*. And they do not refer to locations which makes it simpler to specify environment properties in an implementation–independent way (cf. the specification of `updfst` in section 2.2.1 on page 24 for an application of predicate *disj*):

$$oreach : Object \times Object \times ObjEnv \rightarrow Boolean$$
$$oreach(X, Y, E) \Leftrightarrow \exists L : reach(X, L, E) \wedge Y = obj(L)$$

$$disj : Object \times Object \times ObjEnv \rightarrow Boolean$$
$$disj(X, Y, E) \Leftrightarrow \forall L : \neg reach(X, L, E) \vee \neg reach(Y, L, E)$$

As for predicate *reach*, the basic properties of *disj* are summarized for later reference by a lemma:

**Lemma 3.3 :** Properties of Predicate *disj*

i) $disj(X, Y, E) \wedge \neg reach(X, L, E) \wedge \neg reach(Y, L, E) \Rightarrow disj(X, Y, E\langle L := Z\rangle)$

ii) $disj(X, Y, E) \wedge disj(X, Z, E) \wedge \neg reach(X, L, E) \Rightarrow disj(X, Y, E\langle L := Z\rangle)$

iii) $alive(X, E) \Rightarrow disj(X, new(E, T), E\langle T\rangle)$

iv) $\neg isvoid(X) \wedge typ(X) = dtyp(A) \wedge disj(X, Y, E) \Rightarrow disj(E(loc(A, X)), Y, E)$

□

The predicates *reach*, *oreach*, and *disj* relate objects and locations in one environment. Another important perspective is to look at different environments from one object. We call two environments $E$, $E'$ equivalent w.r.t. an object $X$, denoted by $E \equiv_X E'$, iff $X$ and all objects reachable from $X$ in $E$ or $E'$ have the same state, i.e. they are alive in $E$ iff they are alive in $E'$ and their locations hold the same objects in $E$ and $E'$. We define:

$$\begin{array}{lll} E \equiv_X E' & \Leftrightarrow_{def} & (alive(X,E) \Leftrightarrow alive(X,E')) \\ & \wedge & \forall L : reach(X,L,E) \;\Rightarrow\; E(L) = E'(L) \end{array}$$

According to the env8–axiom, all objects reached from $X$ are alive. That all locations of reachable objects hold the same objects in both environment is a consequence of the second conjunct of the right hand side above and the fact that $X$–equivalence is an equivalence relation. This fact and further interesting properties are summarized by the following lemma:

**Lemma 3.4 :** $X$–Equivalence

i) For any object $X$, $\equiv_X$ is an equivalence relation.

ii) $(\forall X : E \equiv_X E') \;\Rightarrow\; E = E'$

iii) $\neg reach(X,L,E) \;\Rightarrow\; E \equiv_X E\langle L := Y\rangle$

iv) $X \neq new(E, TID) \;\Rightarrow\; E \equiv_X E\langle TID\rangle$

v) $E \equiv_X E' \;\Rightarrow\; (reach(X,L,E) \;\Leftrightarrow\; reach(X,L,E'))$

□

The proof of this lemma is given in appendix A.3.

$X$–equivalence provides a good basis to define further relations on environments. A useful abbreviation is the extension of equivalence from objects to types:

$$E \equiv_T E' \Leftrightarrow_{def} \forall X : \; typ(X) = T \;\Rightarrow\; E \equiv_X E'$$

To specify that methods have no side–effects, the following binary relation on environments $E$, $E'$ plays a prominent role. It expresses that $E'$ differs from $E$ only on objects being alive in $E$:

$$E \ll E' \Leftrightarrow_{def} \forall X : alive(X,E) \;\Rightarrow\; E \equiv_X E'$$

If $E \ll E'$, $E$ is called *less alive* than $E'$. Some interesting properties of the less–alive–relation are summarized by the following lemma:

**Lemma 3.5 :** Less–Aliveness of Environments

i) $\ll$ is a partial order.

ii) $E \ll E' \wedge \mathit{alive}(X,E) \;\Rightarrow\; \mathit{alive}(X,E')$

iii) $E \ll E' \wedge (\mathit{alive}(X,E) \vee \neg\mathit{alive}(X,E')) \;\Rightarrow\; E \equiv_X E'$

iv) $E \ll E' \wedge \neg\mathit{alive}(\mathit{obj}(L),E') \;\Rightarrow\; E \ll E'\langle L := Y\rangle$

v) $E \ll E\langle \mathit{TID}\rangle$

$\square$

## 3.3 Programming Logic

This section presents a programming logic for AICKL. The programming logic is the union of programming–language–independent axioms and rules and the axiomatic semantics of AICKL given in section 3.1.3. It is essentially an adaption and combination of the proof system $G$ and $G_0$ presented in [Apt81]. It allows to prove partial correctness properties for class–based programs. Beside the extension to programs with complex data types, the logic differs from classical Hoare logic in three minor aspects:

1. It has the slightly refined semantics as explained in section 3.1.3.

2. In its presented basic form, it does not support complex expressions and unifies assignments and method calls.

3. It distinguishes between logical and program variables and restricts the occurrence of program variables in pre– and postconditions.

In Hoare logic, program and logical variables are syntactically the same although they behave differently: Substitution (in particular renaming) of and quantification over logical variables in a triple is straightforward (cf. ex-rule and subst-rule below) whereas substitution of program variables is in general not allowed, renaming would affect the program component and its context[8], and quantifying over program variables makes no sense. Temporal logic copes with this different nature of variables by distinguishing between global (i.e. logical) and local variables (cf. e.g. [GU91], p. 233ff.). To stay in the syntax of first-order logic, we represent program variables *syntactically* as constant symbols (cf. end of section 3.1.2). Thus, it is clear from the syntax that renaming of and quantification over program variables is not allowed. In addition to that, allowing only valid program variables to occur in pre- and postconditions can simply be expressed by restricting the signature of pre- and postconditions accordingly.

[8]This is in particular not acceptable in programming environments with proof support.

**Axioms and Rules** The presentation of the logic starts with an axiom and two rules supporting the expected derivations on sequents: the trivial sequent is provable; additional assumptions may be added to a sequent; assumptions may be eliminated if they can be proved from other assumptions:

asumpt-axiom:

$$\mathbf{A} \vdash \mathbf{A}$$

asumpt–intro–rule:

$$\frac{\mathcal{A} \vdash \mathbf{A}}{\mathbf{A}_0 \, , \mathcal{A} \vdash \mathbf{A}}$$

assumpt–elim–rule:

$$\frac{\begin{array}{l}\mathcal{A} \vdash \mathbf{A}_0 \\ \mathbf{A}_0 \, , \mathcal{A} \vdash \mathbf{A}\end{array}}{\mathcal{A} \vdash \mathbf{A}}$$

Together with postcondition weakening (see below), the following axiom states that if the preconditon is false, everything can be shown for the poststate:

false–axiom:

$$\vdash \{ \text{ FALSE } \} \text{ COMP } \{ \text{ FALSE } \}$$

Every property **R** that does not depend on program variables is invariant for all program components. As e.g. { TRUE } COMP { TRUE } doesn't hold in our refined logic (e.g. a statement may try to access an attribute of a void–object), the invariance can not be formulated as an axiom, but has to be expressed as a rule:

inv–rule:

$$\frac{\mathcal{A} \vdash \{ \ \mathbf{P} \ \} \text{ COMP } \{ \ \mathbf{Q} \ \}}{\mathcal{A} \vdash \{ \ \mathbf{P} \wedge \mathbf{R} \ \} \text{ COMP } \{ \ \mathbf{Q} \wedge \mathbf{R} \ \}}$$

where **R** is a $\Sigma$–formula. The following two rules are taken from [Gor88]. As explained in [Apt81], section 3.9, they are not needed to achieve completeness of the proof system in the sense of Cook. However, they are useful for splitting proofs into independent parts and enable to combine method specifications without knowing the body of the methods (in [Apt81] it is assumed that the complete program text is available for proofs which is in general not the case if reasoning is based on interface specifications). E.g. for proving

$$\{\mathbf{P}\} \text{ meth m } \{\mathbf{Q}_1\} \, , \{\mathbf{P}\} \text{ meth m } \{\mathbf{Q}_2\} \vdash \{\mathbf{P}\} \text{ meth m } \{\mathbf{Q}_1 \wedge \mathbf{Q}_2\}$$

without knowledge of the body of m, the following conjunct–rule is indispensable.

conjunct–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}_1\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1\ \} \qquad \mathcal{A} \vdash \{\ \mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_2\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}_1 \wedge \mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1 \wedge \mathbf{Q}_2\ \}}$$

disjunct–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}_1\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1\ \} \qquad \mathcal{A} \vdash \{\ \mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_2\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}_1 \vee \mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1 \vee \mathbf{Q}_2\ \}}$$

The following two rules are classical Hoare rules:

strength–rule:

$$\frac{\mathbf{P}' \Rightarrow \mathbf{P} \qquad \mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}'\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}$$

weak–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \} \qquad \mathbf{Q} \Rightarrow \mathbf{Q}'}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}'\ \}}$$

The distinction between logical and program variables makes the substitution rule very simple:

subst–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[t/Z]\ \}\ \text{COMP}\ \{\ \mathbf{Q}[t/Z]\ \}}$$

where $Z$ is an arbitrary logical variable and $t$ a $\Sigma$–term. The next two rules, the so–called *quantifier rules*, allow to bind logical variables in pre– and postconditions if they do not occur free in the corresponding post– and preconditions. That a variable does not occur free in a formula is expressed by substituting it with a different variable:

all–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}[Y/Z]\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[Y/Z]\ \}\ \text{COMP}\ \{\ \forall Z : \mathbf{Q}\ \}}$$

ex–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}[Y/Z]\ \}}{\mathcal{A} \vdash \{\ \exists Z : \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}[Y/Z]\ \}}$$

where $Z$, $Y$ are arbitrary, but distinct logical variables.

**Proofs and Proof Outlines** This paragraph explains what a proof is, illustrates the application of the above rules and introduces some notations for proofs. To simplify explanations, axioms are considered as rules without antecedents in the following. An *instance* of a rule (or axiom) is obtained by consistently substituting metavariables by formulas, triples, and assumptions. These formulas, triples, and assumptions may contain as well metavariables. A *proof of a sequent* is a (finite) tree the nodes of which are labeled by rule instances such that there is a child for each antecedent $\mathbf{A}$ and the succedent of the child rule equals $\mathbf{A}$. The succedent of the root is the proved sequent. A *proof of a rule* with antecedents $\mathbf{A}_1, .., \mathbf{A}_n$ and succedent $\mathbf{A}_0$ is a proof of $\mathbf{A}_0$ in which the antecedents may occur as axioms.

The typical application of the all–rule is illustrated by the proof of the following derived rule:

$$\frac{\vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\vdash \{\ \forall X : \mathbf{P}\ \}\ \text{COMP}\ \{\ \forall X : \mathbf{Q}\ \}}$$

Of course, this rule is only of interest if $\mathbf{P}$ and $\mathbf{Q}$ contain free occurrences of $X$. The rule can be interpreted as follows: If a triple holds for all $X$ individually, then the postcondition holds for all $X$ as long as the precondition holds for all $X$. This rule is essentially a kind of distributivity law and can be found in many logics. Here is the first–order version:

$$\frac{\vdash \mathbf{P}\ \Rightarrow\ \mathbf{Q}}{\vdash \forall X : \mathbf{P}\ \Rightarrow\ \forall X : \mathbf{Q}}$$

The proof of the triple version of this distributivity rule strengthens the precondition using the valid first–order formula $(\forall X : \mathbf{P})\ \Rightarrow\ \mathbf{P}$; then the all–rule is applied with $X$ for $Z$:

$$\begin{array}{ll}
(\forall X : \mathbf{P})\ \Rightarrow\ \mathbf{P} & \\
\vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \} & \\
\hline
 & [\![\ \text{strength–rule}\ ]\!] \\
\vdash \{\ \forall X : \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \} & \\
\hline
 & [\![\ \text{X not free in } (\forall X : \mathbf{P})\ ]\!] \\
\vdash \{\ (\forall X : \mathbf{P})[Y/X]\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \} & \\
\hline
 & [\![\ \text{all–rule}\ ]\!] \\
\vdash \{\ (\forall X : \mathbf{P})[Y/X]\ \}\ \text{COMP}\ \{\ \forall X : \mathbf{Q}\ \} & \\
\hline
 & [\![\ \text{X not free in } (\forall X : \mathbf{P})\ ]\!] \\
\vdash \{\ \forall X : \mathbf{P}\ \}\ \text{COMP}\ \{\ \forall X : \mathbf{Q}\ \} &
\end{array}$$

For proofs of longer programs the above proof format is very inconvenient. Therefore, we use as well a proof outline technique similar to that proposed in [Owi75]. Proof outlines are annotated programs; they avoid to write the program components again and again. E.g. in the proof outline corresponding to the above proof the component COMP appears only once:

$\{\ \forall X : \mathbf{P}\ \}$
$\Rightarrow\ [\![\ (\forall X : \mathbf{P})$ and $(\forall X : \mathbf{P})[Y/X]$ are syntactically equivalent $]\!]$
$\{\ (\forall X : \mathbf{P})[Y/X]\ \}$
———————————————— $\downarrow$ $[\![$ all–rule $]\!]$
$\{\ (\forall X : \mathbf{P})[Y/X]\ \}$
$\Rightarrow\ [\![\ (\forall X : \mathbf{P})$ and $(\forall X : \mathbf{P})[Y/X]$ are syntactically equivalent $]\!]$
$\{\ (\forall X : \mathbf{P})\ \}$
$\Rightarrow\ [\![$ strengthening with $(\forall X : \mathbf{P}) \Rightarrow \mathbf{P}\ ]\!]$
$\{\ \mathbf{P}\ \}$

```
COMP
```

$\{\ \mathbf{Q}\ \}$
———————————————— $\uparrow$
$\{\ \forall X : \mathbf{Q}\ \}$

Whereas strengthening depends only on preconditions and weakening only on post-conditions, the quantifier rules are related to pre– and postcondition: One of them has to fulfill the constraint of non–free occurrence and the other is modified. Similarly, the substitution and invariance rule modify pre– and postcondition. To relate the corresponding pre– and postconditions in proof outlines, we use lines with up and down arrows as parentheses. Comments on proof steps are included in these $[\![$ double brackets $]\!]$.

The ex–rule is mainly used to eliminate logical variables. To illustrate this, we show that the specifications of the predeclared and implicitly declared methods could have been given as well by two axioms: One axiom specifying the result, one axiom specifying the environment modification. We demonstrate the needed proof technique for the idt–axiom:

$\{\ \text{self} = S\ \}$
$\Rightarrow$
$\{\ \exists E : \text{self} = S \wedge \$ = E\ \}$
———————————— $\downarrow$ $[\![$ ex–rule $]\!]$
$\{\ \text{self} = S \wedge \$ = E\ \}$

```
meth T::idt()
```

$\{\ \text{result} = S \wedge \$ = E\ \}$
$\Rightarrow$
$\{\ \text{result} = S\ \}$
———————————— $\uparrow$
$\{\ \text{result} = S\ \}$

Correspondingly, we can prove $\{\ \$ = E\ \}$ meth T::idt() $\{\ \$ = E\ \}$ . The idt–axiom can be derived from this two–triple–specification by the conjunct–rule. Another application of the ex–rule leads to a more interesting result. Whenever the result of a method and the environment in the poststate are expressed in terms of the parameters and the environment in the prestate, i.e. whenever a method annotation has the following form (to keep things simple, we assume only one explicit parameter) with $t_r(S, P, E)$ and $t_e(S, P, E)$ being $\Sigma$–formulas:

$$\{\ \mathbf{R} \wedge \text{self} = S \wedge \text{p} = P \wedge \$ = E\ \}$$
$$\text{meth T::m}\ \{\ \text{result} = t_r(S, P, E) \wedge \$ = t_e(S, P, E)\ \}$$

we can compute the precondition that is sufficient to establish a given postcondition. For a postcondition **P** of a method annotation, we can show:

$$\{\ \mathbf{R} \wedge \mathbf{P}[t_r(\text{self}, \text{p}, \$)/\text{result}\ ,\ t_e(\text{self}, \text{p}, \$)/\$]\ \}\ \text{meth T::m}\ \{\ \mathbf{P}\ \} \tag{3.3}$$

To prove the latter method annotation from the former let us assume that the logical variables $S, P, E$ do not occur free in **P** (otherwise rename variables and use the subst–rule at the end of the derivation). As **P** is a $\Gamma_{post}$–formula, $\mathbf{P}[t_r(S, P, E)/\text{result}\ ,\ t_e(S, P, E)/\$]$ is a $\Sigma$–formula so that the inv–rule can be applied:

$$\begin{array}{ll}
\{\ \mathbf{R} \wedge \mathbf{P}[t_r(\text{self}, \text{p}, \$)/\text{result}\ ,\ t_e(\text{self}, \text{p}, \$)/\$]\ \} & \\
\Rightarrow & \\
\{\ \exists S, P, Q : \mathbf{R} \wedge \text{self} = S \wedge \text{p} = P \wedge \$ = E & \\
\quad \wedge\ \mathbf{P}[t_r(S, P, E)/\text{result}\ ,\ t_e(S, P, E)/\$]\ \} & \\
\hline
& \downarrow\ \ [\![\ \text{ex–rule}\ ]\!] \\
\{\ \mathbf{R} \wedge \text{self} = S \wedge \text{p} = P \wedge \$ = E & \\
\quad \wedge\ \mathbf{P}[t_r(S, P, E)/\text{result}\ ,\ t_e(S, P, E)/\$]\ \} & \\
\hline
& \downarrow\ \ [\![\ \text{inv–rule}\ ]\!] \\
\{\ \mathbf{R} \wedge \text{self} = S \wedge \text{p} = P \wedge \$ = E\ \} & \\
\texttt{meth T::m} & \\
\{\ \text{result} = t_r(S, P, E) \wedge \$ = t_e(S, P, E)\ \} & \\
\hline
& \uparrow \\
\{\ \text{result} = t_r(S, P, E) \wedge \$ = t_e(S, P, E) & \\
\quad \wedge\ \mathbf{P}[t_r(S, P, E)/\text{result}\ ,\ t_e(S, P, E)/\$]\ \} & \\
\Rightarrow & \\
\{\ \mathbf{P}\ \} & \\
\hline
& \uparrow \\
\{\ \mathbf{P}\ \} &
\end{array}$$

Method annotations of the form shown by 3.3 allow us to handle method call statements in a way similar to assignments in classical Hoare logic; i.e. we can prove

$$\{\ \mathbf{R} \wedge \mathbf{Q}[t_r(\text{E}_0, \text{E}_1, \$)/\text{v}\ ,\ t_e(\text{E}_0, \text{E}_1, \$)/\$]\ \}\ \text{v} := \text{E}_0\ .\ \text{m}(\text{E}_1)\ \{\ \mathbf{Q}\ \} \tag{3.4}$$

for an arbitrary $\Gamma_{body(m\_encl)}$–formula $\mathbf{Q}$ where m_encl is the enclosing method of the call. The proof of 3.4 illustrates the use of call– and var–rule. To keep things simple, we assume that $\mathbf{Q}$ only contains the program variables v, w, and \$ and that $\mathbf{Q}$ and $\mathbf{R}$ do not contain the logical variable $W$:

$$\{\ \mathbf{R} \wedge \mathbf{Q}[t_r(\mathrm{E}_0, \mathrm{E}_1, \$)/\mathrm{v}\ ,\ t_e(\mathrm{E}_0, \mathrm{E}_1, \$)/\$]\ \}$$

$$\Rightarrow\ [\![\ W \text{ does not occur free in } \mathbf{R} \text{ and } \mathbf{Q}\ ]\!]$$

$$\{\ \mathbf{R}[\mathrm{w}/W] \wedge \mathbf{Q}[t_r(\mathrm{E}_0, \mathrm{E}_1, \$)/\mathrm{v}\ ,\ W/\mathrm{w}\ ,\ t_e(\mathrm{E}_0, \mathrm{E}_1, \$)/\$][\mathrm{w}/W]\ \}$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\downarrow \quad [\![\ \text{var–rule}\ ]\!]$$

$$\{\ \mathbf{R} \wedge \mathbf{Q}[t_r(\mathrm{E}_0, \mathrm{E}_1, \$)/\mathrm{v}\ ,\ W/\mathrm{w}\ ,\ t_e(\mathrm{E}_0, \mathrm{E}_1, \$)/\$]\ \}$$

$$\Rightarrow$$

$$\{\ \mathbf{R} \wedge \mathbf{Q}[\mathrm{result}/\mathrm{v}, W/\mathrm{w}][t_r(\mathrm{E}_0, \mathrm{E}_1, \$)/\mathrm{result}\ ,\ t_e(\mathrm{E}_0, \mathrm{E}_1, \$)/\$]\ \}$$

$$\mathrm{v}{:=}\ \mathrm{E}_0\ .\ \mathrm{m}(\mathrm{E}_1) \quad [\![\ \text{using } \mathbf{Q}[\mathrm{result}/\mathrm{v}, W/\mathrm{w}] \text{ for } \mathbf{P} \text{ in } 3.3\ ]\!]$$

$$\{\ \mathbf{Q}[\mathrm{result}/\mathrm{v}, W/\mathrm{w}][\mathrm{v}/\mathrm{result}]\ \}$$

$$\Rightarrow\ [\![\ \text{result does not occur free in } \mathbf{Q}\ ]\!]$$

$$\{\ \mathbf{Q}[W/\mathrm{w}]\ \}$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\uparrow$$

$$\{\ \mathbf{Q}[W/\mathrm{w}][\mathrm{w}/W]\ \}$$

$$\Rightarrow\ [\![\ W \text{ does not occur free in } \mathbf{Q}\ ]\!]$$

$$\{\ \mathbf{Q}\ \}$$

Specializing axiom 3.4 to the predeclared, implicitly declared, read and write methods yields:

| | | |
|---|---|---|
| $\vdash\ \{\ \mathbf{P}[\mathit{bool}(\mathrm{E}_0 = \mathrm{E}_1)/\mathrm{v}]\ \}$ | $\mathrm{v} := \mathrm{E}_0\ .\ \mathrm{equ}(\mathrm{E}_1)$ | $\{\mathbf{P}\}$ |
| $\vdash\ \{\ \mathbf{P}[\mathrm{E}_0/\mathrm{v}]\ \}$ | $\mathrm{v} := \mathrm{E}_0$ | $\{\mathbf{P}\}$ |
| $\vdash\ \{\ \mathbf{P}[\mathit{new}(\$, \mathrm{T})/\mathrm{v}\ ,\ \$\langle \mathrm{T} \rangle/\$]\ \}$ | $\mathrm{v} := \mathrm{E}_0\ .\ \mathrm{new\ T}$ | $\{\mathbf{P}\}$ |
| $\vdash\ \{\ \mathrm{RANGE} \wedge \mathbf{P}[\mathit{int}(\mathit{aI}(\mathrm{E}_0) + \mathit{aI}(\mathrm{E}_1))/\mathrm{v}]\ \}$ | $\mathrm{v} := \mathrm{E}_0\ .\ \mathrm{add}(\mathrm{E}_1)$ | $\{\mathbf{P}\}$ |
| $\vdash\ \{\ \mathbf{P}[\mathit{bool}(\mathit{aI}(\mathrm{E}_0) < \mathit{aI}(\mathrm{E}_1))/\mathrm{v}]\ \}$ | $\mathrm{v} := \mathrm{E}_0\ .\ \mathrm{less}(\mathrm{E}_1)$ | $\{\mathbf{P}\}$ |
| $\vdash\ \{\ \mathbf{P}[\mathit{bool}(\neg \mathit{aB}(\mathrm{E}_0))/\mathrm{v}]\ \}$ | $\mathrm{v} := \mathrm{E}_0\ .\ \mathrm{not}()$ | $\{\mathbf{P}\}$ |
| $\vdash\ \{\ \neg \mathit{isvoid}(\mathrm{E}_0) \wedge \mathit{typ}(\mathrm{E}_0) = \mathrm{T} \wedge \mathbf{P}[\$(\mathrm{E}_0.\mathrm{att})/\mathrm{v}]\ \}$ | $\mathrm{v} := \mathrm{E}_0\ .\ \mathrm{T{::}att}$ | $\{\mathbf{P}\}$ |
| $\vdash\ \{\ \neg \mathit{isvoid}(\mathrm{E}_0) \wedge \mathit{typ}(\mathrm{E}_0) = \mathrm{T} \wedge \mathbf{P}[\$\langle \mathrm{E}_0.\mathrm{att} := \mathrm{E}_1 \rangle/\$]\ \}$ | $\mathrm{E}_0\ .\ \mathrm{T{::}att} := \mathrm{E}_1$ | $\{\mathbf{P}\}$ |

where T is the type of expression $\mathrm{E}_0$ and RANGE abbreviates $\mathit{minint} \leq \mathit{aI}(\mathrm{E}_0) + \mathit{aI}(\mathrm{E}_1) \leq \mathit{maxint}$. We refer to these axioms by *mid*–call–ax where *mid* is the identifier of the corresponding method; in particular the assignment axiom is named idt–call–ax. Thus, the weakest precondition technique can be applied for these methods to automate proofs.

To demonstrate an application of the axioms derived above and the interplay between programming logic and state properties, we prove that a newly created object is disjoint from all living objects:

$$\vdash \{\ \mathit{alive}(Y,\$)\ \}\ \mathrm{v} := \mathrm{new}\ \mathrm{T}\ \{\ \mathit{disj}(Y,\mathrm{v},\$)\ \}$$

Instantiating **P** in new–call–ax by $\mathit{disj}(Y,\mathrm{v},\$)$, it remains to show that $\mathit{alive}(Y,\$)$ implies $\mathit{disj}(Y,\mathit{new}(\$,\mathrm{T}),\$\langle\mathrm{T}\rangle)$ which is lemma 3.3.(iii).

## 3.4 Metarules: Typing and Invariants

This section is about three basic properties that hold for all AICKL programs. It shows that AICKL programs are type safe, that variables only hold living objects, and that programs can only modify locations reachable from their parameters and local variables.

**Type Safety** Analyzing typing properties of programming languages, we distinguish between type correctness and type safety. A program is called *type correct* if it satisfies the typing rules of the underlying programming language. E.g. if type T has an integer attribute att and v is a variable of type T, the statement `v.att := 7` is type correct w.r.t. to the rules of AICKL. We call a program component *type safe*, if the execution of the component starting in a well–typed state leads to a well–typed state upon termination. A state is called *well–typed* if all variables and parameters are well–typed and the object environment is well–typed. A parameter or variable v is called well–typed in a state if $\mathit{typ}(\mathrm{v}) = \mathrm{T}$ where T is the type declared for v. An object environment $E$ is called well–typed ( denoted by $\mathit{wt}(E)$ ) if all locations $L$ are well–typed, i.e. $\mathit{typ}(E(L)) = \mathit{ltyp}(L)$. According to this definition the statement `v.att := 7` is not type safe, because v may hold the constant void(T) in which case the semantics of AICKL permits an arbitrary behaviour, in particular termination in an ill–typed state. Thus, type correctness does not guarantee type safety. And type safety can assert type correctness only in case of terminating executions[9].

In this thesis, we consider only a restricted version of type safety. This restricted version expresses type safety relative to specifications of program components: If the triple $\{\ \mathbf{R}\ \}$ COMP $\{$ TRUE $\}$ can be proved for a component COMP, then we can show that well–typed prestates satisfying **R** lead to well–typed poststates upon termination, i.e. we can show $\{\ \mathbf{R} \wedge \mathbf{TA}_{pre}\ \}$ COMP $\{\ \mathbf{TA}_{post}\ \}$ where the type annotations $\mathbf{TA}_{pre}$ and $\mathbf{TA}_{post}$ express that pre– and poststate are well–typed. These type annotations are made precise by the following definition:

**Definition 3.6 :** Adding Type Annotations
Let **A** be a triple; *adding type annotations* to **A** means to add type information about the parameters and variables to the pre- and postcondition of **A** and to require that the environment is well–typed. The resulting triple is denoted by $\mathit{typed}(\mathbf{A})$. The type annotation of a triple **A** depends on its kind:

[9] A program that loops may be type safe although it contains ill–typed statements following the loop.

- If **A** is a method annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ meth T::m $\{\ \mathbf{Q}\ \}$, *typed*(**A**) is

  $$\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} typ(p_i) = \mathrm{TP}_i \wedge wt(\$)\ \}$$

  ```
  T::m
  ```

  $$\{\ \mathbf{Q} \wedge typ(\mathrm{result}) = \mathrm{TR} \wedge wt(\$)\ \}$$

  where $p_i$, $i = 0, \ldots, y$, are the formal parameters of T::m with types $\mathrm{TP}_i$ and TR denotes the result type of T::m.

- If **A** is a statement annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ STAT $\{\ \mathbf{Q}\ \}$, *typed*(**A**) is

  $$\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} typ(p_i) = \mathrm{TP}_i \wedge \bigwedge_{i=0}^{z} typ(v_i) = \mathrm{TV}_i \wedge wt(\$)\ \}$$

  ```
  STAT
  ```

  $$\{\ \mathbf{Q} \wedge \bigwedge_{i=0}^{y} typ(p_i) = \mathrm{TP}_i \wedge \bigwedge_{i=0}^{z} typ(v_i) = \mathrm{TV}_i \wedge wt(\$)\ \}$$

  where $p_i$, $i = 0, \ldots, y$, are the formal parameters and $v_i$, $i = 0, \ldots, z$ are the local variables of the enclosing method with types $\mathrm{TP}_i$ and $\mathrm{TV}_i$ respectively.

The *typed*–operator is canonically extended to sets $\mathcal{A}$ of triples.

□

**Lemma 3.7 :** Type Safety of AICKL
If there exists a proof for $\mathcal{A} \vdash \mathbf{A}$, then $typed(\mathcal{A}) \vdash typed(\mathbf{A})$ can be proved.

□

**Proof**
The proof runs by induction on the depth of the proof tree for $\mathcal{A} \vdash \mathbf{A}$. As this is a proof techique used several times in this thesis, we go into some detail here.

**Induction Base:** For the induction base, we have to show that the type annotations can be added to all axioms:

- assumpt–axiom: If $\mathbf{A} \vdash \mathbf{A}$ is an axiom, then $typed(\mathbf{A}) \vdash typed(\mathbf{A})$ is an axiom.

- false–axiom: Let $\mathbf{TA}_{pre}$ and $\mathbf{TA}_{post}$ denote the type annotation for the pre– and poststate of COMP:

  $\{\ \mathrm{FALSE} \wedge \mathbf{TA}_{pre}\ \}$
  $\Rightarrow$
  $\{\ \mathrm{FALSE}\ \}$
  COMP $[\![$ false–axiom $]\!]$
  $\{\ \mathrm{FALSE}\ \}$
  $\Rightarrow$
  $\{\ \mathrm{FALSE} \wedge \mathbf{TA}_{post}\ \}$

- equ–axiom:

$$
\begin{array}{l}
\quad \{\ \mathrm{self} = S \wedge \mathrm{p} = P \wedge \$ = E \wedge \mathit{typ}(\mathrm{self}) = \mathrm{T} \wedge \mathit{typ}(\mathrm{p}) = \mathrm{T} \wedge \mathit{wt}(\$)\ \} \\
\quad \Rightarrow \\
\quad \{\ \mathrm{self} = S \wedge \mathrm{p} = P \wedge \$ = E \wedge \mathit{wt}(E)\ \} \\
\quad \overline{\hspace{20em}}\downarrow \quad [\![\ \text{inv–rule}\ ]\!] \\
\quad \{\ \mathrm{self} = S \wedge \mathrm{p} = P \wedge \$ = E\ \} \\
\text{meth T::equ(p): BOOL} \\
\quad \{\ \mathrm{result} = \mathit{bool}(S = P) \wedge \$ = E\ \} \\
\quad \overline{\hspace{20em}}\uparrow \\
\quad \{\ \mathrm{result} = \mathit{bool}(S = P) \wedge \$ = E \wedge \mathit{wt}(E)\ \} \\
\quad \Rightarrow \quad [\![\ \text{definition of } \mathit{typ}\ ]\!] \\
\quad \{\ \mathrm{result} = \mathit{bool}(S{=}P) \wedge \$ = E \wedge \mathit{typ}(\mathit{bool}(S{=}P)) = \mathit{BOOL} \wedge \mathit{wt}(E)\ \} \\
\quad \Rightarrow \\
\quad \{\ \mathrm{result} = \mathit{bool}(S == P) \wedge \$ = E \wedge \mathit{typ}(\mathrm{result}) = \mathit{BOOL} \wedge \mathit{wt}(\$)\ \}
\end{array}
$$

- idt–axiom:

$$
\begin{array}{l}
\quad \{\ \mathrm{self} = S \wedge \$ = E \wedge \mathit{typ}(\mathrm{self}) = \mathrm{T} \wedge \mathit{wt}(\$)\ \} \\
\quad \Rightarrow \\
\quad \{\ \mathrm{self} = S \wedge \$ = E \wedge \mathit{typ}(S) = \mathrm{T} \wedge \mathit{wt}(E)\ \} \\
\quad \overline{\hspace{20em}}\downarrow \quad [\![\ \text{inv–rule}\ ]\!] \\
\quad \{\ \mathrm{self} = S \wedge \$ = E\ \} \\
\text{meth T::idt(): T} \\
\quad \{\ \mathrm{result} = S \wedge \$ = E\ \} \\
\quad \overline{\hspace{20em}}\uparrow \\
\quad \{\ \mathrm{result} = S \wedge \$ = E \wedge \mathit{typ}(S) = \mathrm{T} \wedge \mathit{wt}(E)\ \} \\
\quad \Rightarrow \\
\quad \{\ \mathrm{result} = S \wedge \$ = E \wedge \mathit{typ}(\mathrm{result}) = \mathrm{T} \wedge \mathit{wt}(\$)\ \}
\end{array}
$$

- new–axiom:

$$
\begin{array}{l}
\quad \{\ \$ = E \wedge \mathit{typ}(\text{self}) = \text{T} \wedge \mathit{wt}(\$)\ \} \\
\quad \Rightarrow \\
\quad \{\ \$ = E \wedge \mathit{wt}(E)\ \} \\
\quad \overline{\hspace{12cm}} \downarrow \quad [\![\ \text{inv–rule}\ ]\!] \\
\quad \{\ \$ = E\ \} \\
\text{meth T::new(): T} \\
\quad \{\ \text{result} = \mathit{new}(E, \text{T}) \wedge \$ = E\langle \text{T} \rangle\ \} \\
\quad \overline{\hspace{12cm}} \uparrow \\
\quad \{\ \text{result} = \mathit{new}(E, \text{T}) \wedge \$ = E\langle \text{T} \rangle \wedge \forall L : \mathit{typ}(E(L)) = \mathit{ltyp}(L)\ \} \\
\quad \Rightarrow\ [\![\ \text{env5–axiom}\ ]\!] \\
\quad \{\ \text{result} = \mathit{new}(E, \text{T}) \wedge \$ = E\langle \text{T} \rangle \wedge \forall L : \mathit{typ}(E\langle \text{T} \rangle(L)) = \mathit{ltyp}(L)\ \} \\
\quad \Rightarrow\ [\![\ \text{env11–axiom}\ ]\!] \\
\quad \{\ \text{result} = \mathit{new}(E, \text{T}) \wedge \$ = E\langle \text{T} \rangle \wedge \mathit{typ}(\text{result}) = \text{T} \wedge \mathit{wt}(\$)\ \}
\end{array}
$$

- Proving type annotations for add–axiom, less–axiom, and not–axiom is similar to the above derivations.

- read–att–axiom: Using the proof steps as above, it remains to show the following for read–att–axiom:

$$\neg \mathit{isvoid}(S) \wedge \mathit{typ}(S) = \text{T} \wedge \mathit{wt}(E) \ \Rightarrow\ \mathit{typ}(E(S.\text{T::att})) = \text{RT}_{att}$$

  where $\text{RT}_{att}$ denotes the range type of attribute att. The premise implies the existence of an object identifier *OID* with $S = \mathit{object}(\text{T}, \mathit{OID})$:

$$
\begin{array}{ll}
 & \mathit{typ}(E(S.\text{T::att})) \\
= & \quad [\![\ \mathit{wt}(E) \text{ and expansion of } S\ ]\!] \\
 & \mathit{ltyp}(\mathit{object}(\text{T}, \mathit{OID}).\text{T::att}) \\
= & \quad [\![\ \text{expansion of abbreviation } \_.\_\ ]\!] \\
 & \mathit{ltyp}(\mathit{loc}(\mathit{object}(\text{T}, \mathit{OID}), \text{T::att})) \\
= & \quad [\![\ \text{definition of } \mathit{loc} \text{ and definition of } \mathit{ltyp}\ ]\!] \\
 & \mathit{rtyp}(\text{T::att}) \\
= & \quad [\![\ \text{definition of } \mathit{rtyp}\ ]\!] \\
 & \text{RT}_{att}
\end{array}
$$

- write–att–axiom: Using the proof steps as above, it remains to show the following for write–att–axiom:

$$
\begin{array}{l}
\neg \mathit{isvoid}(S) \wedge \mathit{typ}(S) = \text{T} \wedge \mathit{typ}(P) = \text{RT}_{att} \wedge \mathit{wt}(E) \\
\qquad \Rightarrow\ \forall L : E\langle S.\text{T::att} := P \rangle(L) = \mathit{ltyp}(L)
\end{array}
$$

where $\mathrm{RT}_{att}$ denotes the range type of attribute att. The premise implies the existence of an object identifier $OID$ with $S = object(\mathrm{T}, OID)$. In particular, we have $obj(S.\mathrm{T}{::}\mathrm{att}) = S$. To show the conclusion, we distinguish whether $L$ is equal or unequal to $S.\mathrm{T}{::}\mathrm{att}$. For the latter case, the conclusion can be obtained from the env1–axiom. For the former case, we distinguish the subcases: 1. $alive(S, E) \wedge alive(P, E)$. 2. $\neg alive(S, E)$. 3. $\neg alive(P, E)$.

1. Subcase: $alive(S, E) \wedge alive(P, E)$.

$$\begin{array}{cl} & typ(E\langle S.\mathrm{T}{::}\mathrm{att} := P\rangle(S.\mathrm{T}{::}\mathrm{att}) \\ = & \quad [\![ \text{ assumption of subcase 1 and env2 } ]\!] \\ & typ(P) \\ = & \quad [\![ \text{ premise from above } ]\!] \\ & \mathrm{RT}_{att} \\ = & \quad [\![ \text{ as shown in the derivation for read–att–axiom } ]\!] \\ & ltyp(S.\mathrm{T}{::}\mathrm{att}) \end{array}$$

2. Subcase: $\neg alive(S, E)$.

$$\begin{array}{cl} & typ(E\langle S.\mathrm{T}{::}\mathrm{att} := P\rangle(S.\mathrm{T}{::}\mathrm{att}) \\ = & \quad [\![ \text{ assumption of subcase 2 together with env6 and env3 } ]\!] \\ & typ(init(ltyp(S.\mathrm{T}{::}\mathrm{att}))) \\ = & \quad [\![ \; typ(init(T)) = T \; ]\!] \\ & ltyp(S.\mathrm{T}{::}\mathrm{att}) \end{array}$$

3. Subcase: $\neg alive(P, E)$.

$$\begin{array}{cl} & typ(E\langle S.\mathrm{T}{::}\mathrm{att} := P\rangle(S.\mathrm{T}{::}\mathrm{att}) \\ = & \quad [\![ \text{ assumption of subcase 3 and env4 } ]\!] \\ & typ(E(S.\mathrm{T}{::}\mathrm{att})) \\ = & \quad [\![ \; wt(E) \; ]\!] \\ & ltyp(S.\mathrm{T}{::}\mathrm{att}) \end{array}$$

**Induction Step:** For the induction step, we may assume that an instantiation of a rule is given, say

$$\frac{\begin{array}{c}\mathcal{A} \vdash \mathbf{A}_1 \\ \mathcal{A} \vdash \mathbf{A}_2\end{array}}{\mathcal{A} \vdash \mathbf{A}_0}$$

and that $typed(\mathcal{A}) \vdash typed(\mathbf{A}_i)$ is proven for $i = 1, 2$. We have to derive $typed(\mathcal{A}) \vdash typed(\mathbf{A}_0)$. This is simple for all rules except for the call– and rec–rule:

- call–rule: As the assumptions occurring in the instantiations are not needed, we can use the proof outline technique. Let $\mathrm{pv}_i$ $i = 0, \ldots, z$ denote the formal parameters and local variables of the method enclosing the call with types $\mathrm{TPV}_i$, let v = $\mathrm{pv}_0$, and let $R_i$ denote fresh logical variables. Then, the typed annotation for the call can be derived as follows:

$$\{\ \mathbf{P}[\mathrm{E}_0/\mathrm{self}, \mathrm{E}_1/\mathrm{p}_1, \ldots, \mathrm{E}_x/\mathrm{p}_x] \wedge \bigwedge_{i=0}^{z} \mathit{typ}(\mathrm{pv}_i) = \mathrm{TPV}_i \wedge \mathit{wt}(\$)\ \}$$

$\Rightarrow$ ⟦ if variables are well–typed, then expressions are well–typed ⟧

$$\{\ \mathbf{P}[\mathrm{E}_0/\mathrm{self}, \mathrm{E}_1/\mathrm{p}_1, \ldots, \mathrm{E}_x/\mathrm{p}_x] \wedge \bigwedge_{i=0}^{x} \mathit{typ}(\mathrm{E}_i) = \mathrm{TP}_i \wedge \bigwedge_{i=1}^{z} \mathit{typ}(\mathrm{pv}_i) = \mathrm{TPV}_i \wedge \mathit{wt}(\$)\ \}$$

———————— ↓ ⟦ var–rule ⟧

$$\{\ \mathbf{P}[\mathrm{E}_0/\mathrm{self}, \mathrm{E}_1/\mathrm{p}_1, \ldots, \mathrm{E}_x/\mathrm{p}_x] \wedge \bigwedge_{i=0}^{x} \mathit{typ}(\mathrm{E}_i) = \mathrm{TP}_i \wedge \bigwedge_{i=1}^{z} \mathit{typ}(R_i) = \mathrm{TPV}_i \wedge \mathit{wt}(\$)\ \}$$

———————— ↓ ⟦ inv–rule ⟧

$$\{\ \mathbf{P}[\mathrm{E}_0/\mathrm{self}, \mathrm{E}_1/\mathrm{p}_1, \ldots, \mathrm{E}_x/\mathrm{p}_x] \wedge \bigwedge_{i=0}^{x} \mathit{typ}(\mathrm{E}_i) = \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \}$$

v := $\mathrm{E}_0$ . m($\mathrm{E}_1, \ldots, \mathrm{E}_x$) ⟦ applying the call–rule to the ind. hypo. ⟧

$$\{\ \mathbf{Q}[\mathrm{v}/\mathrm{result}] \wedge \mathit{typ}(\mathrm{v}) = \mathrm{TR} \wedge \mathit{wt}(\$)\ \}$$

———————— ↑

$$\{\ \mathbf{Q}[\mathrm{v}/\mathrm{result}] \wedge \mathit{typ}(\mathrm{v}) = \mathrm{TR} \wedge \bigwedge_{i=1}^{z} \mathit{typ}(R_i) = \mathrm{TPV}_i \wedge \mathit{wt}(\$)\ \}$$

———————— ↑

$$\{\ \mathbf{Q}[\mathrm{v}/\mathrm{result}] \wedge \mathit{typ}(\mathrm{v}) = \mathrm{TR} \wedge \bigwedge_{i=1}^{z} \mathit{typ}(\mathrm{pv}_i) = \mathrm{TPV}_i \wedge \mathit{wt}(\$)\ \}$$

$\Rightarrow$ ⟦ type rules of AICKL imply: result type of m equals type of v ⟧

$$\{\ \mathbf{Q}[\mathrm{v}/\mathrm{result}] \wedge \bigwedge_{i=0}^{z} \mathit{typ}(\mathrm{pv}_i) = \mathrm{TPV}_i \wedge \mathit{wt}(\$)\ \}$$

- rec–rule: The derivation starts with the induction hypothesis; then it simplifies the precondition of the consequent and weakens the postcondition; finally, the recursion rule is applied:

$$\frac{\dfrac{\begin{array}{l}\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} \mathit{typ}(\mathrm{p}_i) = \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \} \\ \quad \text{meth T::m}(\mathrm{p}_1, \ldots, \mathrm{p}_z)\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathrm{result}) = \mathrm{TR} \wedge \mathit{wt}(\$)\ \}, \mathcal{A} \\ \vdash \{\ \mathbf{P} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(\mathrm{p}_i) = \mathrm{TP}_i \wedge \bigwedge_{i=1}^{z} \mathit{typ}(\mathrm{v}_i) = \mathrm{TV}_i \wedge \mathit{wt}(\$) \\ \quad \wedge \bigwedge_i \mathrm{v}_i = \mathit{init}(\mathrm{TV}_i)\ \} \\ \text{BODY(T::m)}\ \{\ \mathbf{Q} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(\mathrm{p}_i) = \mathrm{TP}_i \wedge \bigwedge_{i=1}^{z} \mathit{typ}(\mathrm{v}_i) = \mathrm{TV}_i \wedge \mathit{wt}(\$)\ \}\end{array}}{\begin{array}{l}\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} \mathit{typ}(\mathrm{p}_i) = \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \} \\ \quad \text{meth T::m}(\mathrm{p}_1, \ldots, \mathrm{p}_z)\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathrm{result}) = \mathrm{TR} \wedge \mathit{wt}(\$)\ \}, \mathcal{A} \\ \vdash \{\ \mathbf{P} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(\mathrm{p}_i) = \mathrm{TP}_i \wedge \mathit{wt}(\$) \wedge \bigwedge_{i=1}^{z} \mathrm{v}_i = \mathit{init}(\mathrm{TV}_i)\ \} \\ \quad \text{BODY(T::m)}\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathrm{result}) = \mathrm{TR} \wedge \mathit{wt}(\$)\ \}\end{array}}}{\begin{array}{l}\mathcal{A} \vdash \{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} \mathit{typ}(\mathrm{p}_i) = \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \} \\ \quad \text{meth T::m}(\mathrm{p}_1, \ldots, \mathrm{p}_z)\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathrm{result}) = \mathrm{TR} \wedge \mathit{wt}(\$)\ \}\end{array}}$$

**end of proof**

**Liveness Properties** Similar to type safety, we show that all objects held by program variables are alive. As non–static objects can be referenced in programs only through program variables and as objects can only reference living objects (according to env8–axiom), all objects reachable in a program state are alive. We define liveness annotations for methods and statements. Liveness annotations can be implicitly carried around in proofs, just as we have shown for type annotations.

**Definition 3.8 :** Adding Liveness Annotations
Let **A** be a triple; *adding liveness annotations* to **A** means to add liveness information about the parameters and variables to the pre- and postcondition of **A**. The resulting triple is denoted by *pv_alive*(**A**). The liveness information of a triple **A** depends on its kind:

- If **A** is a method annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ meth T::m $\{\ \mathbf{Q}\ \}$, *pv_alive*(**A**) is

  $$\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} alive(\mathrm{p}_i, \$)\ \}$$
  `T::m`
  $$\{\ \mathbf{Q} \wedge alive(\mathrm{result}, \$)\ \}$$

  where $\mathrm{p}_i$, $i = 0, \ldots, y$, are the formal parameters of T::m.

- If **A** is a statement annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ STAT $\{\ \mathbf{Q}\ \}$, *pv_alive*(**A**) is

  $$\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} alive(\mathrm{p}_i, \$) \wedge \bigwedge_{i=0}^{z} alive(\mathrm{v}_i, \$)\ \}$$
  `STAT`
  $$\{\ \mathbf{Q} \wedge \bigwedge_{i=0}^{y} alive(\mathrm{p}_i, \$) \wedge \bigwedge_{i=0}^{z} alive(\mathrm{v}_i, \$)\ \}$$

  where $\mathrm{p}_i$, $i = 0, \ldots, y$, are the formal parameters and $v_i$, $i = 0, \ldots, z$ are the local variables of the method enclosing STAT.

The *pv_alive*–operator is canonically extended to sets $\mathcal{A}$ of triples.

□

**Lemma 3.9 :** Live–Closeness of AICKL
If there exists a proof for $\mathcal{A} \vdash \mathbf{A}$, then $pv\_alive(\mathcal{A}) \vdash pv\_alive(\mathbf{A})$ can be proved.

□

Objects cannot be deleted, i.e. all operations on object environments leave living objects alive. This invariance of liveness carries over to program annotations. I.e. if we add $alive(X, \$)$ as conjunct to the precondition, we can show that it holds in the postcondition. The *all_alive*–operator is used to denote this extension of pre– and postcondition. Based on this operator, the invariance of liveness can be stated as follows:

**Lemma 3.10 :** Invariance of Liveness
If there exists a proof for $\mathcal{A} \vdash \mathbf{A}$, then $\mathit{all\_alive}(\mathcal{A}) \vdash \mathit{all\_alive}(\mathbf{A})$ can be proved.
□

The proofs of the two lemmas above run by induction on the depth of the proof tree for $\mathcal{A} \vdash \mathbf{A}$. In most of their steps, they directly follow the proof of type safety. As they do not contribute new derivation aspects, they are not presented.

**Local Updating Property** Similar to type safety and the liveness properties, we can show that methods only update locations that are reachable from actual parameters or that belong to objects created by the method. This so–called *local updating property* can e.g. be used to prove that properties of objects disjoint from method parameters remain invariant under method execution. We first formulate the local updating property and then illustrate its application.

**Definition 3.11 :** Adding Non–Reachable Location Annotations
Let $\mathbf{A}$ be a triple; *adding annotations for non–reachable locations* to $\mathbf{A}$ means to add a conjunct to the pre- and postcondition of $\mathbf{A}$ stating the following: If a location $L$ belonging to a living object is not reachable from the variables, it holds the same object in pre– and poststate. The resulting triple is denoted by $\mathit{nrloc}(\mathbf{A})$. The precise form of the conjunct depends on the kind of the triple $\mathbf{A}$:

- If $\mathbf{A}$ is a method annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ meth T::m $\{\ \mathbf{Q}\ \}$, $\mathit{nrloc}(\mathbf{A})$ is

  $$\{\ \mathbf{P} \wedge \$(L) = X \wedge \mathit{alive}(\mathit{obj}(L), \$) \wedge \textstyle\bigwedge_{i=0}^{y} \neg \mathit{reach}(\mathrm{p}_i, L, \$)\ \}$$

  ```
  T::m
  ```

  $$\{\ \mathbf{Q} \wedge \$(L) = X \wedge \mathit{alive}(\mathit{obj}(L), \$) \wedge \neg \mathit{reach}(\mathrm{result}, L, \$)\ \}$$

  where $\mathrm{p}_i$, $i = 0, \ldots, y$, are the formal parameters of T::m and $L$ and $X$ are logical variables not occurring in $\mathbf{A}$.

- If $\mathbf{A}$ is a statement annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ STAT $\{\ \mathbf{Q}\ \}$, $\mathit{nrloc}(\mathbf{A})$ is

  $$\{\ \mathbf{P} \wedge \$(L) = X \wedge \mathit{alive}(\mathit{obj}(L), \$) \wedge \textstyle\bigwedge_{i=0}^{y} \neg \mathit{reach}(\mathrm{p}_i, L, \$) \wedge \bigwedge_{i=0}^{z} \neg \mathit{reach}(\mathrm{v}_i, L, \$)\ \}$$

  ```
  STAT
  ```

  $$\{\ \mathbf{Q} \wedge \$(L) = X \wedge \mathit{alive}(\mathit{obj}(L), \$) \wedge \textstyle\bigwedge_{i=0}^{y} \neg \mathit{reach}(\mathrm{p}_i, L, \$) \wedge \bigwedge_{i=0}^{z} \neg \mathit{reach}(\mathrm{v}_i, L, \$)\ \}$$

  where $\mathrm{p}_i$, $i = 0, \ldots, y$, are the formal parameters and $v_i$, $i = 0, \ldots, z$ are the local variables of the method enclosing STAT and $L$ and $X$ are logical variables not occurring in $\mathbf{A}$.

The *nrloc*–operator is canonically extended to sets $\mathcal{A}$ of triples.
□

**Lemma 3.12 :** Local Updating Property
If there exists a proof for $\mathcal{A} \vdash \mathbf{A}$, then $nrloc(\mathcal{A}) \vdash nrloc(\mathbf{A})$ can be proved.

$\square$

**Proof**
The proof runs by induction on the depth of the proof tree for $\mathcal{A} \vdash \mathbf{A}$. Here we consider only the interesting axioms; i.e. the axioms dealing with the object environment. All other cases can be proved as demonstrated for the type annotations.

- new–axiom:

$$\begin{array}{l} \quad \{\ \$ = E \wedge \$(L) = X \wedge alive(obj(L), \$) \wedge \neg reach(\text{self}, L, \$)\ \} \\ \text{meth T::new(): T} \quad [\![\ \text{from new–axiom with lemma 3.10}\ ]\!] \\ \quad \{\ \text{result} = new(E, \text{T}) \wedge \$ = E\langle \text{T} \rangle \wedge E(L) = X \\ \quad\quad \wedge\ alive(obj(L), \$) \wedge \neg reach(\text{result}, L, E)\ \} \\ \quad \Rightarrow \quad [\![\ \text{env5–axiom and lemma 3.2.(v)}\ ]\!] \\ \quad \{\ \text{result} = new(E, \text{T}) \wedge \$ = E\langle \text{T} \rangle \wedge \$(L) = X \\ \quad\quad \wedge\ alive(obj(L), \$) \wedge \neg reach(\text{result}, L, \$)\ \} \end{array}$$

- read–att–axiom: Essentially, we have to prove:

$$\neg isvoid(S) \wedge typ(S) = \text{T} \wedge \neg reach(S, L, E) \ \Rightarrow\ \neg reach(E(S.\text{T::att}), L, E)$$

  And this is a simple consequence from the definition of *reach*.

- write–att–axiom: Essentially, we have to prove:

$$\begin{array}{l} \neg isvoid(S) \wedge typ(S) = \text{T} \wedge \neg reach(S, L, E) \wedge \neg reach(P, L, E) \\ \quad \Rightarrow\ E\langle S.\text{T::att} := P \rangle(L) = E(L) \end{array}$$

  From $\neg reach(S, L, E)$ we get $L \neq S.\text{T::att}$; thus, env1–axiom yields the equation in the conclusion.

**end of proof**

In program verification, it is often more convenient to work with the following less general lemma:

**Lemma 3.13 :** Equivalence Property
If there exists a proof for $\vdash \{\ \mathbf{P}\ \}$ meth T::m $\{\ \mathbf{Q}\ \}$, then the following triple can be proved:

$$\vdash \{\ \mathbf{P} \wedge \$ = E \wedge alive(X, \$) \wedge \bigwedge_{i=0}^{y} disj(X, \text{p}_i, \$)\ \}\ \text{meth T::m}\ \{\ \mathbf{Q} \wedge \$ \equiv_X E\ \}$$

$\square$

The proof is given in appendix A.4.

Lemma 3.13 enables to prove that certain properties are invariant under method execution. These proofs can be carried out without knowing the method body. To demonstrate this, let $p$ be some predicate with signature:

$$p : \mathit{Object} \times \mathit{ObjEnv} \rightarrow \mathit{Boolean}$$

Predicate $p(X, E)$ expresses that some property holds for $X$ in environment $E$. In order to apply the above lemma this property may only depend on those parts of $E$ that are reachable from $X$; i.e. $p$ has to satisfy:

$$p(X, E) \wedge E \equiv_X E' \;\Rightarrow\; p(X, E') \tag{3.5}$$

For program specification and verification, requirement 3.5 is not a restriction. In particular, this requirement is met by all environment–depending predicates used in this thesis (see e.g. lemma 3.4.(v)). Based on lemma 3.13 we can prove

$$\{\ \mathbf{R} \wedge p(X,\$) \wedge \mathit{alive}(X,\$) \wedge \textstyle\bigwedge_{i=0}^{y} \mathit{disj}(X, \mathrm{p}_i, \$)\ \}$$

`T::m`

$$\{\ p(X,\$)\ \}$$

from $\{\ \mathbf{R}\ \}$ meth T::m $\{$ TRUE $\}$ where $\mathrm{p}_i$, $i = 0, \ldots, y$, are the formal parameters of T::m. We derive:

$$\{\ \mathbf{R} \wedge p(X,\$) \wedge \mathit{alive}(X,\$) \wedge \textstyle\bigwedge_{i=0}^{y} \mathit{disj}(X, \mathrm{p}_i, \$)\ \}$$

$$\Rightarrow$$

$$\{\ \exists E : \mathbf{R} \wedge p(X,E) \wedge \$ = E \wedge \mathit{alive}(X,\$) \wedge \textstyle\bigwedge_{i=0}^{y} \mathit{disj}(X, \mathrm{p}_i, \$)\ \}$$

$$\overline{\hspace{20em}}\downarrow \quad [\![\ \text{ex–rule}\ ]\!]$$

$$\{\ \mathbf{R} \wedge p(X,E) \wedge \$ = E \wedge \mathit{alive}(X,\$) \wedge \textstyle\bigwedge_{i=0}^{y} \mathit{disj}(X, \mathrm{p}_i, \$)\ \}$$

T::m $[\![$ lemma 3.13 and inv–rule $]\!]$

$$\{\ p(X,E) \wedge \$ \equiv_X E\ \}$$

$$\Rightarrow \quad [\![\ 3.5\ ]\!]$$

$$\{\ p(X,\$)\ \}$$

$$\overline{\hspace{20em}}\uparrow$$

$$\{\ p(X,\$)\ \}$$

The generalization of this result to predicates with several objects as arguments and additional arguments is straightforward.

# Chapter 4

# Specifying and Verifying Object–Oriented Programs

This chapter defines a small object–oriented programming language by combining and extending the program constructs described in chapter 2 and 3. The programming logic for this language is described as an extension of the logic in chapter 3. With this formal background, interface specification techniques are reviewed and refined. The meaning of interface specifications is defined as sets of triples. In particular, the meaning of invariants and their role for verification is discussed. The fundamental patterns needed for program verification in our logic are introduced and illustrated by extending the list example of chapter 2.

## 4.1 Object–Oriented Programs

This section defines a small object–oriented programming language with subtyping and encapsulation that is based on interfaces and classes as described in chapter 2 and 3. A program is essentially a set of interfaces implemented by a set of classes. In order to discuss modularization concepts, we distinguish between imported and declared interfaces. Accordingly, the central task of programming is to produce a set of interfaces and their implementation using a given set of *imported* interfaces. I.e. programming means to produce libraries for use in other programs; in particular, the designation of a main class or a main method is neglected. The following sections will show how this view to programming affects and is influenced by program specification and verification.

This section is structured similar to section 3.1. It introduces syntax, object environment, and programming logic for the object–oriented extension of the kernel language. In addition, it proves central properties of the language.

### 4.1.1 A Kernel Language with Subtyping and Encapsulation

Based on AICKL, we define a new language AICKL* with subtyping, encapsulation, and a primitive module concept. Subtyping essentially means that variables of type T may hold objects of subtypes of T. Encapsulation is a well known technique for structuring programs and hiding information at the interface between clients and implementations of data types. In the context of specification and verification, encapsulation turns out to be as well important to give class invariants an adequate meaning (see section 4.2). The primitive module concept allows to simulate program extensions, i.e. the use of already declared interfaces for the declaration of new interfaces. In order to keep it as simple as possible a module may only import one other module. This way, we avoid naming and consistency problems between imported modules and constructs for renaming.

The following three subsections provide the syntactical definition of the language AICKL*, explain it by examples, and discuss design decisions and the relation to other languages. Definition and explanation were separated in order to have the definition in a more compact form for later reference.

#### 4.1.1.1 The Form of AICKL* Programs

An AICKL* program is a (finite) set of modules totally ordered by the import relation; i.e. there is one module that imports no module and all other modules import exactly one module. The former module is called the *predeclared* module. The other modules are called *declared*. The module that is not imported by any other module is called the *top* module of a program.

An AICKL* module is a (finite) set of interfaces and a (finite) set of classes. The interfaces are partially ordered according to the subtype relation. Interfaces and classes are essentially as defined in section 2.2 and 3.1 respectively. The changes, extensions, and the interplay between interfaces and classes is defined in the following. An interface is either *abstract* or *concrete*. For each module, an interface is either *public* or *private*. These notions for interfaces are used as well for the types and methods declared by the interfaces; in particular, types declared by concrete interfaces are called *concrete types*.

A module M2 importing module M1, declaring abstract interfaces $TA_1, \ldots, TA_a$ and concrete interfaces $TC_1, \ldots, TC_c$, and exporting interfaces $TE_1, \ldots, TE_e$ is written with the following syntax[1]:

```
module  M2  import M1  is
   abstract interface  TA_1    ...  end
      ...
   abstract interface  TA_a    ...  end
   concrete interface  TC_1    ...  end
      ...
   concrete interface  TC_c    ...  end
   class  TC_1    ...  end
      ...
   class  TC_c    ...  end
   public  TE_1, ... ,TE_e
end
```

[1]The order in which interfaces are given is arbitrary, i.e. abstract and concrete interfaces may alternate.

The public clause at the end of the module declares which of the interfaces of M2 are *public*; interfaces not being declared public are private. A module M2 importing a module M1 imports exactly the public interfaces of M1. Correspondingly, we say that a module *exports* its public interfaces.

The following context conditions apply for a program with modules $\mathrm{M}_0, \ldots, \mathrm{M}_n$ where $\mathrm{M}_0$ denotes the predeclared module and $\mathrm{M}_{i+1}$ imports $\mathrm{M}_i$:

1. All interfaces are subtypes of OBJECT.
2. For any $\mathrm{M}_i$, the union of the sets of imported and declared interfaces is well–formed[2].
3. For any $\mathrm{M}_i$, the set of public interfaces has to be a subset of the union of imported and declared interfaces.
4. For any $\mathrm{M}_i$, the set of all public interfaces is well–formed.
5. Concrete interfaces are minimal w.r.t. the subtype ordering; i.e. concrete types have no subtypes.
6. For each concrete interface TC in a declared module there is exactly one class TC and vice versa. The methods of the concrete interface form a subset of the methods given in the class interface.

According to these context conditions, only concrete interfaces have implementations. Thus, each object is of a concrete type (the predeclared types BOOL and INT are as well concrete; see below). If we view a type as the set of its objects, an abstract type is the union of all objects of its subtypes. This view is helpful to understand the meaning of abstract and concrete types and interfaces. It will be formally defined in subsection 4.1.2.

The predeclared module STD_MOD provides the interfaces for OBJECT, BOOL, and INT and declares at least the following (the `include` directive is explained in section 2.3.1):

```
module  STD_MOD  is
   abstract interface  OBJECT  is
      meth  equ( p: OBJECT ): BOOL
      meth  idt(): SAME
   end

   concrete interface  BOOL  subtype of OBJECT  is
      include  OBJECT
      meth  not(): BOOL
   end
```

[2]Cf. section 2.3.1 on page 31 for the definition of well–formedness.

```
    concrete interface  INT  subtype of OBJECT  is
       include  OBJECT
       meth  add ( i: INT ): INT
       meth  less( i: INT ): BOOL
    end
    public  OBJECT, BOOL, INT
 end
```

As in AICKL, the methods equ and idt express the test on equality and the identity (cf. section 3.1, page 37). The requirement that all interfaces are subtypes of OBJECT guarantees that all interfaces contain the methods equ and idt. To meet the subtype condition, the explicit parameter of equ has type OBJECT in any interface, i.e. method equ is slightly more general in AICKL* than in AICKL: it can compare objects of different types.

Subtyping occurs at two points in the definition of AICKL* : As a relation between interfaces (cf. definition 2.1) and to redefine typing constraints for method calls. In AICKL, types of actual parameters and results have to match exactly their formal counterparts. In AICKL*, this context condition is weakened as explained with the following method call statement:

v := $\mathrm{EXP}_0$ . m( $\mathrm{EXP}_1$,...,$\mathrm{EXP}_z$)

We assume that the call statement occurs in a module named M and that T denotes the type of $\mathrm{EXP}_0$. If T is a concrete type declared in M, its *class* interface must contain a method with identifier m and the correct number of parameters. If T is an abstract or imported type, the interface of T must contain such a method. The types of actual parameter expressions $\mathrm{EXP}_i$ must be subtypes of m's formal parameters. The type of variable v must be a supertype of the result type of m.

Beside regular methods, AICKL* supports *cast methods.* Cast methods are implicitly declared and considered to be global; they are not associated with interfaces. For each type T, there is a cast method having the same name. The self–parameter of a cast method is of type OBJECT, the result is of type T:

```
meth  OBJECT::T(): T
```

Casts are used to narrow the type of an expression: The cast method T returns the given self–object *XS* if the type of *XS* is a subtype of T; otherwise the behaviour of a cast is unspecified (cf. section 4.1.3). In order to exclude cast statements that always behave in an unspecified way, we require that the type of the argument expression in a cast T has to be a supertype of T. In addition to the ordinary syntax for method calls, casts may as well be written as follows:

```
v :=  (T) EXP
```

The expressions of AICKL* are the same as in AICKL except that *void(T)* is only legal if *T* is a concrete type (cf. section 4.1.2).

#### 4.1.1.2 Explaining AICKL*

Compared to what was already introduced in chapters 2 and 3, AICKL* supports three additional concepts: subtyping, encapsulation, and a primitive module concept that allows to distinguish between imported and declared interfaces. We will demonstrate these three concepts in reverse order.

**Modularization Aspects** The module concept of AICKL* allows only a linear modularization of programs; i.e. a program is a finite chain of modules $\mathrm{M}_0, ..., \mathrm{M}_{top}$ where $\mathrm{M}_0$ is module STD_MOD and module $\mathrm{M}_{i+1}$ imports module $\mathrm{M}_i$. Extending a program $\mathcal{P}$ means to add a new module that imports $\mathcal{P}$'s top module. This primitive module concept is certainly not sufficient for practical use, but it is appropriate for our needs: It provides a simple framework to express program extensions and to discuss specification and verification of "open" programs. To illustrate program extensions, we assume the following module ARRAY_MOD:

```
module  ARRAY_MOD  import  STD_MOD  is
   concrete interface  ARRAY  subtype of OBJECT  is
      include  OBJECT
      meth  create( size: INT ): ARRAY
      meth  get( ix: INT ): OBJECT
      meth  set( ix: INT, x: OBJECT )
      meth  size(): INT
      meth  resize( size: INT ): ARRAY
   end
    ...
   public ARRAY, OBJECT, INT, BOOL
end
```

In a first step, we use ARRAY_MOD to implement a module LIST_MOD that provides an abstract interface for lists and two different list implementations:

```
module  LIST_MOD  import ARRAY_MOD  is
   abstract interface  LIST   subtype of OBJECT  is  ...  end
   concrete interface  CLIST  subtype of LIST    is  ...  end
   concrete interface  PLIST  subtype of LIST    is  ...  end

   class  CLIST  is  ...  end
   class  PLIST  is  ...  end

   public LIST, CLIST, PLIST, OBJECT, INT, BOOL
end
```

The interfaces for ARRAY, OBJECT, BOOL, and INT are given above. Interfaces LIST and CLIST are given in chapter 2, class CLIST was presented in section 3.1.1. Interface and implementation of PLIST are presented below. To provide another example for program extensions, we add a new tiny module to the above program containing an interface with a sorting method for lists:

```
module  SORT_MOD  import LIST_MOD  is
   concrete interface  SORT   subtype of OBJECT  is
      include  OBJECT
      meth  sort( l: LIST ): LIST
   end
   class  SORT  is  ...  end

   public SORT, LIST, CLIST, PLIST, OBJECT, INT, BOOL
end
```

The extended program "imports" the public interfaces of LIST_MOD; the interface ARRAY and implementations are hidden. The import and export behaviour of the above program is summarized by the following figure:

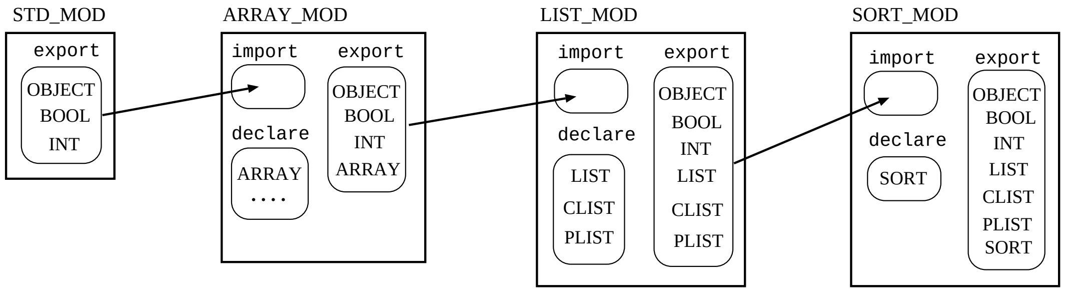


Figure 4.1: Export and import behaviour of the example modules

**Encapsulation** AICKL* supports two mechanisms to express encapsulation: 1. Interfaces can be hidden within modules. E.g. interface ARRAY is private in module LIST_MOD, because it is not declared public. Private interfaces of a module M may not be used in modules importing M. 2. Methods defined in a class C can be hidden. This is done by omitting them in the concrete interface corresponding to C. Typically, read and write methods for attributes and auxiliary methods are hidden. As an example, we consider the interface PLIST that is a subtype of LIST and additionally enables to iterate through lists and selectively update elements. There are methods to initialize the iterator, to check whether the iterator is defined, to step to the next list element, to get the element of the current position, and to set the element at the current position to a given object:

```
concrete interface  PLIST  subtype of LIST  is
   include LIST
   meth  init()
   meth  isdef(): BOOL
   meth  next()
   meth  get(): INT
   meth  set( x: INT )
end
```

The interface PLIST is implemented by a class with three attributes. The first attribute references an array for storing the list elements. The second attribute records the size of the list, the third attribute records the current position of the iterator:

```
class  PLIST  is
   attr  arr: ARRAY
   attr  siz: INT
   attr  pos: INT
   ...
end
```

Thus, the interface for PLIST hides the read and write methods for the three attributes and the method new. Of course, user declared methods can be hidden by this technique as well (see next paragraph). Many other object–oriented programming languages use keywords like "private" to hide methods. This way, the programmer need not write down the public (concrete) interface of the class. We prefer the more verbose technique here, because it simplifies and clarifies the context conditions.

**Subtyping** In a language with subtyping, a variable or parameter can hold objects of different types at different points of execution. E.g. a variable lv of type LIST can hold objects of type CLIST and type PLIST, because CLIST and PLIST are subtypes of type LIST. A call of method m on a variable of an abstract type is handled as follows: Determine the[3] concrete type T of the object held by the variable and execute the code for method T::m. The subtyping rules guarantee that such a method exists. E.g. in a call lv.isempty(), either PLIST::isempty or CLIST::isempty are executed depending on the current object held by lv. Thus, the method executed for a call statement is dynamically selected (cf. section 2.3.1). The semantics of method selection is further investigated in section 4.1.3.

If an algorithm only depends on an interface shared by a number of data structures, subtyping allows to formulate this algorithms once based on the shared interface. We illustrate this by an implementation of the method sort declared in module SORT_MOD. Sorting a list $L$ is done by recursively sorting the rest of $L$ and inserting the first element:

```
class  SORT  is
   meth  sort( l: LIST ): LIST
      lv: LIST; ev: INT; bv: BOOL;
      bv := l.isempty()
      if  bv  then   result := l
      else
         lv :=  l.rest();
         ev :=  l.first();
         lv :=  sort( lv );
         result :=  sorted_ins( ev, lv )
      end
   end
```

[3] Each object is of exactly one concrete type; cf. p. 75 .

```
      meth  sorted_ins( e: INT; l: List ): List
        //  inserts e in sorted list l
      end
   end
```

In addition to subtyping, this class illustrates some other aspects: 1. In connection with the concrete interface for SORT given above, it exemplifies the encapsulation of auxiliary methods, here of the method sorted_ins. 2. It demonstrates how classes without attributes can be used to implement ordinary procedures.

AICKL* supports multiple subtyping, i.e. an interface can be subtype of two or more interfaces. The following module MULTIPLE_SUBSORT serves as an example:

```
module  MULTIPLE_SUBSORT  import STD_MOD  is
   abstract interface  PRINT    subtype of OBJECT  is
      include  OBJECT
      meth  print()
   end
   abstract interface  BROWSE   subtype of OBJECT  is
      include  OBJECT
      meth  browse()
   end
   abstract interface  ANIMATE  subtype of PRINT, BROWSE  is
      include  PRINT
      meth  browse()
      meth  animate()
   end
   public  PRINT, BROWSE, ANIMATE, OBJECT, INT, BOOL
end
```

Objects of type PRINT can be printed, objects of type BROWSE provide a browse method. As interface ANIMATE is a subtype of PRINT and BROWSE, objects of type ANIMATE are printable and support browsing. In addition to this they can be animated. Module MULTIPLE_SUBSORT illustrates in particular how subtyping can be used to structure behavioural properties of objects.

#### 4.1.1.3 Discussing AICKL*

The strongly–typed, imperative object–oriented programming languages form a fairly large and practical important language class including C++, Java, Eiffel, Sather, Oberon, Modula-3, Simula, and BETA. AICKL* is a kernel language in that language class, in the sense that

- it covers the central concepts and constructs of languages in that class (at least as far as specification and verification is concerned) and
- all techniques developed for AICKL* are applicable to the other languages of that class.

This subsection discusses the design of AICKL* in relation to other object–oriented languages. It focuses on three aspects: 1. Encapsulation and modularization. 2. Inheritance. 3. Type casting.

**Encapsulation and Modularization** Encapsulation and modularization is very differently realized in existing object–oriented programming languages. Some languages (e.g. BETA) do not support encapsulation of data or hiding of methods on the class level, but only on the module level. Others provide almost no modularization constructs beyond classes (like e.g. C++). Thus, there are no kernel constructs for encapsulation and modularization that are common to all of these languages. Nevertheless, both concepts are essential for the specification and verification of object–oriented programs:

- Without encapsulation, a user of an interface can manipulate data representations in an arbitrary way. Thus, typical invariants of data representations cannot be guaranteed by the interface (e.g. that singly linked lists are acyclic). Consequently, proof obligations for verifying integrity constraints for data representations would spread all over the program whereas with encapsulation they can be concentrated in the verification of the interface specification.

- Modularization is needed to structure programs into parts that are imported and parts that are declared within the program itself. Such a structuring is essential to analyze the effect that importing of interfaces into a larger program context has for the validity of interface specifications. Subsection 4.2.3.2 illustrates by an example that properties holding for a given set of interfaces may become invalid if the set of interfaces are used within a larger program.

In AICKL*, methods listed in interfaces are public. Methods that are declared in a class, but not listed in the corresponding concrete interface are not accessible outside the module. But such methods can be used within the enclosing module (cf. the context condition on method calls in subsection 4.1.1.1). The reason for this design decision is that an interface is often implemented with the help of auxiliary classes. E.g. a doubly linked list implementation needs two classes, one for list heads and one for list elements. Methods of such classes need mutual access. This restricted form of privacy corresponds to attribute declarations without modifier in Java. In terms of C++, such methods are private, but are friend methods for all classes of the enclosing module. An attribute in AICKL* is public if read and write methods are public; it is read–only if only the read method is public; it is private if read and write methods are private.

Similar to Java, AICKL* enables to declare interfaces to be private. Declaring an interface to be private makes all its methods and the type name inaccessible outside the module. The role of private interfaces in the context of specification and verification is shortly discussed in subsection 4.2.3.3. Further encapsulation constructs can be found in languages supporting subclassing (cf. the next paragraph for a discussion about subclassing). These constructs (e.g. accessibility kind `protected` in C++)

determine how accessibility properties are inherited from super– to subclasses. As AICKL* does not support subclassing, such constructs could be omitted.

As already explained above, the modularization construct in AICKL* is kept as primitive as possible and only introduced to have a clean notation to talk about program extensions. Every realistic module concept supports at least its primitive capabilities.

**Inheritance** Inheritance in object–oriented programming languages has two aspects, subtyping and code reuse. Subtyping corresponds to the users[4] view to a type: A subtype can be used in all places where the supertype is allowed; a supertype can be used to express algorithms that work for all its subtypes. Code reuse is interesting for implementors. They want to reuse code of existing implementations for the implementation of new types. This could be done by copy, paste, and modify, but a safer and better structured way is certainly desirable. In many object–oriented programming languages subtyping and code reuse are tightly coupled; we call this coupling of subtyping and code reuse *subclassing*. Subclassing is a mechanism to construct a new class SUBC from a given class C such that SUBC and C are related as follows:

- The interface of SUBC is a subtype of the interface of C in the sense of definition 2.1 or of a restricted version thereof.
- SUBC *inherits* the attributes from C, i.e. SUBC–objects have at least the attributes of C–objects.
- SUBC *inherits* the method implementations from C, i.e. by default SUBC possesses at least the (public) methods of C with the same implementation[5]. To change the default, inherited methods can be modified (cf. [MMPN93]) or overwritten by other method implementations.
- In addition to the inherited attributes and methods, SUBC may contain further attributes and methods.

In its pure form as sketched above, subclassing assumes that the implementation of a subtype is similar to the implementation of the supertype. As this is in general not true (consider e.g. the type of all objects possessing a print method; cf. subsection 4.1.1.2), most object–oriented languages support so–called *abstract classes*. A class is abstract if some of its methods have no implementation; i.e. an abstract class is a mixture between an interface and a class. Existing object–oriented programming languages provide fairly different solutions to the problem how to combine subclassing, subtyping, and mechanisms for code reuse. We consider three examples:

1. In C++, subtyping can only be introduced via subclassing. C++ supports abstract classes, and enables to derive a subclass from several superclasses; this

[4]By the term "*user of type T*" we refer to persons using T, e.g. to implement other types.

[5]Thus, the program code for these methods is reused and executable code need to include only one copy of the code for these methods.

mechanism is often called multiple inheritance (in our terminology multiple subclassing would be more appropriate).

2. Java supports (single) subclassing as described above (keyword `extends`) and, in addition to that, multiple subtyping declared by the so–called "implements"–relation. I.e. code reuse is only possible along a tree–like hierarchy whereas the subtype relation may be an arbitrary partial order.

3. Sather completely separates subtyping and code reuse. In particular, it does not support subclassing. Code reuse is described by additional language constructs and can be done as well between classes that are not comparable in the subtype ordering. Sather enables to declare new types to be supertypes of existing types.

AICKL* is similar to Sather in that it does not support subclassing. Analyzing the different realizations of subclassing in existing object–oriented programming languages, we found that the common kernel of these realizations corresponds to copying code from the super– to the subclass. Other aspects of subclassing are very language–specific and difficult to formalize in a general way. Often these aspects are defined by an operational semantics that makes it difficult to derive properties of the subclass from properties of the superclass (appendix A.5 illustrates these problems by a small Java example). A more detailed discussion about the relation of subtyping and code reuse can be found in [SOM94].

**Type Casting** Almost every object–oriented language provides constructs to narrow the type of an expression. Typical constructs for this purpose are type case statements and type casts. A type case statement is a case statement that branches according to the type of the object resulting from an expression evaluation. A type cast[6] checks whether the object resulting from the evaluation of an expression is of a certain type; if this is the case, normal execution continues where the type information gained from the cast may be exploited; otherwise execution aborts.

The possibility to narrow types is in particular necessary to simulate parametric polymorphism by subtyping. The canonical example is a list class the elements of which are of the universal type OBJECT. We can use this class to manage lists of any type T; but each time we select an element of the list, we have to narrow the type from OBJECT to T. Otherwise we could not apply methods of type T to the selected element.

As explained at the end of subsection 4.1.1.1, type casts are realized in AICKL* by so–called cast methods. Using methods to express casts enables us to keep the programming logic of AICKL unchanged. We only have to add a specification for cast methods (cf. section 4.1.3). The reason why cast methods are not associated with interfaces becomes clear if one tries to treat them as regular methods: Let us assume a module with types OBJECT, $T_1$,.., $T_n$ and some subtype ordering. We

[6]Languages with subclassing often support another kind of type casts that map objects of a subclass to "corresponding" objects of the superclass. These casts widen the type of the object and modify its behaviour (e.g. in C++ attributes occurring only in the subclass are eliminated; in Java hidden attributes are uncovered).

need casts from OBJECT to each of the $T_i$'s. As the result type of a method cannot be parameterized, interface OBJECT would have to contain $n$ methods for casting. The context condition that each type is a subtype of OBJECT and the requirement resulting from the subtype relation (definition 2.1) enforces that each interface has $n$ methods for casting. In particular, there are methods casting objects from one concrete type to another which is certainly undesirable. Even worse, adding a new type to a program would enlarge the interfaces of all other types. These consequences refrained us from unifying regular and cast methods.

### 4.1.2 Data and State Model for AICKL*

The data and state model for AICKL* is essentially the same as that for AICKL (cf. section 3.1.2). The only difference is concerned with the modeling of types and their relations. We have to incorporate abstract types, have to express the subtype relation, and have to model aspects of the module concept.

Just as we assumed a sort *TypId* for type identifiers of classes/concrete interfaces, we assume a sort *ATypId* for type identifiers of abstract interfaces and adapt the definition of *Type* correspondingly (cf. section 3.1.2):

**data type**

$$\begin{array}{lcl} Type & = & BOOL \\ & \mid & INT \\ & \mid & ct(\,TypId\,) \\ & \mid & at(\,ATypId\,) \end{array}$$

**end data type**

The definitions of *Object* and *Location* remain unchanged; i.e. there are only objects of concrete type and the function *typ* yields for each object its concrete type.

The subtype relation expresses a partial order on *Type* and is denoted by $\preceq$. Beside the following program–independent axioms, there are axioms specifying the subtype relation between the types declared in a program under consideration. This specification can e.g. be done by an axiom for each pair of types T1, T2 specifying whether one is a subtype of the other or whether they are incomparable. Then, extending a program means for the subtype relation that a finite number of axioms has to be added specifying the subtype relation for the newly declared types.

$$T \preceq T \tag{4.1}$$

$$T \preceq T' \wedge T' \preceq T'' \;\Rightarrow\; T \preceq T'' \tag{4.2}$$

$$T \preceq T' \wedge T' \preceq T \;\Rightarrow\; T = T' \tag{4.3}$$

$$T \preceq ct(TID) \;\Rightarrow\; T = ct(TID) \tag{4.4}$$

All predicate and function definitions given for the data and state model without subtyping carry over to the case with subtyping except for predicate $T$–equivalence and well–typing of object environments. The later predicates are adapted as follows:

$$E \equiv_T E' \Leftrightarrow_{def} \forall X : \; typ(X) \preceq T \;\Rightarrow\; E \equiv_X E'$$

$$wt(E) \;\Leftrightarrow\; _{def} \forall L : \; typ(E(L)) \preceq ltyp(L)$$

**Module Aspects** A new aspect of AICKL* compared to AICKL is the module concept and — related to it — private interfaces. In order to make these aspects available for specification and verification, we have to provide a formalization for them. We give here a fairly primitive formalization that demonstrates how these aspects can be expressed and what role they play in specification and verification. This formalization was developed for the needs in the rest of this thesis. It is not meant to solve the formalization problem for more realistic module concepts.

We assume a sort *ModId* that contains module identifiers. Module identifiers are ordered according to the import relation denoted by $\sqsupset$, i.e. if module M1 imports module M2, we write $M1 \sqsupset M2$. The import relation is a partial order on *ModId* and can be axiomatized similar to the subtype relation above. We call a type T extern to a module M, denoted by $\mathit{extern}(\mathrm{T},\mathrm{M})$, if it is not declared within M or within modules directly or indirectly imported by M. $\mathit{extern}(\mathrm{T},\mathrm{M})$ can be axiomatized by enumerating the non–extern types. In particular, we assume that we can derive:

$$\mathrm{M1} \sqsupset \mathrm{M0} \wedge \mathit{extern}(\mathrm{T},\mathrm{M1}) \;\Rightarrow\; \mathit{extern}(\mathrm{T},\mathrm{M0})$$

Recall from above that an interface TPRIV is private in a module M, if it is imported by or declared in M but not declared public. I.e. type TPRIV may not be used in modules importing M. In particular, type TPRIV may not occur in an attribute declaration outside M. A comprehensive axiomatization for private and public interfaces presupposes a formalization of static module properties like attribute A or interface T is declared in module M. This can be modeled similar to the relation between attributes and types which is expressed by the functions *dtyp* and *rtyp*. For the purposes of this thesis, we do not need so much detail. Instead we assume that the following formula can be derived for all attributes A:

$$\mathit{rtyp}(\mathrm{A}) = \mathrm{TPRIV} \;\Rightarrow\; \neg\mathit{extern}(\mathit{dtyp}(\mathrm{A}),\mathrm{M}) \tag{4.5}$$

In addition to this, we assume that private types are closed. A type T is called *closed* if it is not allowed to add further subtypes to T or one of its subtypes. I.e. if a type is closed all its subtypes are closed. The fact that a type T with direct subtypes $\mathrm{T}_1,\ldots,\mathrm{T}_s$ is closed is defined as follows:

$$\mathit{closed}(\mathrm{T}) \;\Leftrightarrow_{\mathit{def}}\; \bigwedge_{i=1}^{s} \mathit{closed}(\mathrm{T}_i) \;\wedge\; \forall S:\; S \prec \mathrm{T} \;\Leftrightarrow\; \bigvee_{i=1}^{s} S \preceq \mathrm{T}_i$$

In particular, concrete types are closed. Closing a type means to restrict possible extensions of the program. We assume that declaring an interface to be private closes the corresponding type. As it is illustrated in the following section, closing types simplifies verification. Thus, it should be possible for programmers to close types without declaring them private. In AICKL*, a type can be closed by prefixing its interface with the keyword `closed`. (For more implementation–oriented reasons, Java allows to close a class; this is done by the keyword `final`; cf. 8.1.2.2 in [JG96], p. 133.)

### 4.1.3 Axiomatic Semantics for AICKL*

The axiomatic semantics for AICKL* is a proper extension of AICKL's semantics (see chapter 3). Only one modification is needed concerning the type assumptions in the recursion rule. With subtyping, we may only assume that the type of an object held by a local variable at the beginning of method execution is a subtype of the type of the variable:

rec*–rule:

$$\frac{\begin{array}{l}\{\ \mathbf{P}\ \}\ \ \text{meth T::m}(\mathrm{p}_1,\ldots,\mathrm{p}_z)\ \{\ \mathbf{Q}\ \}\ ,\mathcal{A}\\ \qquad\vdash\ \{\ \mathbf{P}\wedge\bigwedge_i \mathit{typ}(\mathrm{v}_i)\preceq \mathrm{TV}_i\wedge\bigwedge_i \mathit{static}(\mathrm{v}_i)\ \}\ \text{BODY(T::m)}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A}\vdash\ \{\ \mathbf{P}\ \}\ \ \text{meth T::m}(\mathrm{p}_1,\ldots,\mathrm{p}_z)\ \{\ \mathbf{Q}\ \}}$$

The axiomatic semantics of AICKL* has to capture two new language constructs: cast methods and abstract methods, i.e. methods of abstract interfaces. Cast methods are axiomatized similar to the identity methods. They require the self–object to satisfy the corresponding type constraint:

cast–axiom:

$$\vdash\ \{\ \mathit{typ}(\text{self})\preceq \mathrm{T}\ \}\ \ \text{meth OBJECT::T()}\ \ \{\ \text{result}=\text{self}\hat{}\ \wedge\ \$=\$\hat{}\ \}$$

More interesting is the proof rule that enables to verify annotations of abstract methods. We derive this rule from an informal semantics of the dynamic selection mechanism for methods. Let T be an abstract type, $\mathrm{T}_1,\ldots,\mathrm{T}_k$ its direct subtypes, and m be one of its methods. The basic idea is simple: To prove something for T::m, we have to prove it for all the corresponding methods $\mathrm{T}_i$::m in the subtypes. Two aspects need some attention: 1. What are the precise type assumptions about the parameters? 2. How are program extensions handled?

In order to get the type assumptions right, we implement the abstract method T::m by some pseudo–code and derive the subtype–rule with the programming logic developed so far. The pseudo–code performs a case distinction according to the type of the self–object. Depending on the type, the self–object is casted to one of T's subtypes and the corresponding method associated with the subtype is called. To keep things simple, we assume in the example that T has only the two subtypes T1 and T2, and only one parameter:

```
 meth  T::m( p: TP ): TR is
    v1: T1 ;  v2: T2 ;
    if $typ$(self) $\preceq$ T1  then
            v1     :=  (T1) self ;
            result :=  v1.m(p)
    else if $typ$(self) $\preceq$ T2  then
            v2     :=  (T2) self ;
            result :=  v2.m(p)
    else
            abort
    end end
end
```

If we apply the programming logic to prove $\{\ \mathbf{R}\ \}$ meth T::m $\{\ \mathbf{Q}\ \}$, we get the following proof obligations:

$$\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1} \wedge \mathbf{R}\ \}\ \text{meth T1::m}\ \{\ \mathbf{Q}\ \}$$
$$\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T2} \wedge \mathbf{R}\ \}\ \text{meth T2::m}\ \{\ \mathbf{Q}\ \}$$
$$\{\ \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T1} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T2} \wedge \mathbf{R}\ \}\ \text{abort}\ \{\ \mathbf{Q}\ \}$$

The only axiom that can be used to prove the last triple is the false–axiom, but for arbitrary **R** it is not applicable. If we don't care about program extension, we can use the knowledge that T1 and T2 are the only subtypes of T, i.e.

$$\mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \;\Rightarrow\; \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1} \vee \mathit{typ}(\mathrm{self}) \preceq \mathrm{T2} \tag{4.6}$$

If we substitute $\mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathbf{P}$ for **R** and apply the above implication, the precondition for the abort statement becomes false so that this triple can be proved independent of **P**. Because of $\mathit{typ}(\mathrm{self}) \preceq \mathrm{T}i \;\Rightarrow\; \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}$, we can abbreviate our derivation by the following rule:

$$\frac{\begin{array}{l}\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1} \wedge \mathbf{P}\ \}\ \text{meth T1::m}\ \{\ \mathbf{Q}\ \}\\ \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T2} \wedge \mathbf{P}\ \}\ \text{meth T2::m}\ \{\ \mathbf{Q}\ \}\end{array}}{\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathbf{P}\ \}\ \text{meth T::m}\ \{\ \mathbf{Q}\ \}}$$

And this rule can be generalized to abstract types with an arbitrary number of subtypes. But, what does the rule mean if an abstract type has no subtypes at all? This leads us to the second interesting aspect about the subtyping rule: The question how program extensions should be handled. Before we show a technique to deal with program extensions, we illustrate the source of possible problems by a small example:

```
module MOD1  import STD_MOD  is
   abstract interface  T   subtype of OBJECT  is
      include OBJECT
      meth  someconstant(): INT
   end
```

```
    concrete interface  T1  subtype of T  is
       include T
    end
    class  T1  is
       meth  someconstant(): INT  is
          result := 3
       end
    end
    public  T, T1, OBJECT, BOOL, INT
 end
```

The method annotation { TRUE } meth T::someconstant { result = 3 } can be proved with the above subtyping–rule. Now we add the following concrete interface to the program:

```
 module  MOD2  import MOD1  is
    concrete interface  T2  subtype of T  is
       include T
    end
    class  T2  is
       meth  someconstant(): INT  is
          result := 777
       end
    end
    public   T, T1, T2, OBJECT, BOOL, INT
 end
```

This extension obviously invalidates the triple given above. There are essentially two choices to avoid such situations: 1. We can consider the extended program as a new program and prove everything for this program from scratch. 2. While proving properties of the original program we can collect proof obligations that have to be met by all extensions. Whereas the first choice is simpler from a logical point of view (the above subtyping–rule would be sufficient), it is certainly not what is desirable from a practical point of view. In the first choice, the problem of abstract types without subtypes would disappear by definition: If only complete programs are considered, it does not make sense to consider abstract types without subtypes, because there would be no objects for such types. Consequently, one would forbid such situations by a context condition. On the other hand if we allow program extensions with the aim of reusing the properties and proofs of the original program, it makes sense to consider abstract types without subtypes, because subtypes and implementations may be added later.

To handle extensions of programs, we weaken the above rule by adding an assumption to the conclusion. This assumption keeps track of proof obligations for later added interfaces. As an aside, the use of assumptions allows us to avoid an arbitrary number of antecedents in the subtype rule. In summary, we get the following compact form for the subtype–rule:

subtype–rule:

$$\frac{\begin{array}{l}\mathrm{T}' \preceq \mathrm{T} \\ \mathcal{A} \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}' \wedge \mathbf{P}\ \}\ \mathrm{meth\ T'::m}\ \{\ \mathbf{Q}\ \}\end{array}}{\begin{array}{l}\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T}' \wedge \mathbf{P}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathbf{Q}\ \}, \mathcal{A} \\ \ \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathbf{P}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathbf{Q}\ \}\end{array}}$$

This rule provides some flexibility in proving properties about abstract methods. The assumption can be weakened in a stepwise way and it is not necessary to use only properties of methods in direct subtypes for this purpose. We illustrate the subtype–rule with the interfaces T, T1, and T2 from above proving { TRUE } meth T::someconstant { result > 2 }. The first application of the subtype–rule yields:

$$\frac{\begin{array}{l}\mathrm{T1} \preceq \mathrm{T} \\ \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1}\ \}\ \mathrm{meth\ T1::m}\ \{\ \mathrm{result} > 2\ \}\end{array}}{\begin{array}{l}\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T1}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathrm{result} > 2\ \} \\ \ \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathrm{result} > 2\ \}\end{array}}$$

Similarly we derive for T2:

$$\frac{\begin{array}{l}\mathrm{T2} \preceq \mathrm{T} \\ \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T2} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T1}\ \}\ \mathrm{meth\ T2::m}\ \{\ \mathrm{result} > 2\ \}\end{array}}{\begin{array}{l}\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T2} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T1}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathrm{result} > 2\ \} \\ \quad \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T1}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathrm{result} > 2\ \}\end{array}} \ [\![\ \text{subtype–rule}\ ]\!]$$

The consequent of the derived sequent equals the assumption of the sequent derived first. Thus, the assumption elimination rule gives:

$$\begin{array}{l}\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T2} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T1}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathrm{result} > 2\ \} \\ \quad \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathrm{result} > 2\ \}\end{array}$$

Subtype assumptions can be eliminated by closing the corresponding type (cf. section 4.1.2 for the notion of closed types). For example, closing type T would provide property 4.6. By this formula, we can falsify the precondition of the assumption in the sequent derived above and thus eliminate the assumption using false– and assumpt–elim–rule.

For the implicitly declared methods equ and idt, we do not have to care about program extensions, because they always behave in a predefined way. Thus, the subtype–rule would unnecessarily complicate derivations. Consequently, we allow the use of the equ–axiom and idt-axiom as well for abstract types. The situation is similar for the read and write methods and the method new, except that the occurrences of the type in the preconditions have to be substituted by *typ*(self) and some type conditions have to be added. But, as these methods are rarely used for abstract types, we keep the restriction that the type variable in these axioms ranges only over *TypId*.

These considerations finish the description of the semantics of AICKL*. For better reference, the complete programming logic for AICKL* is summarized in appendix A.2. The next section demonstrates the first applications of the logic and some practical matters concerned with typing and invariant properties.

## Typing and Invariant Properties

In section 3.4, we investigated type safety and general invariant properties for AICKL programs. These properties carry over to AICKL*. We show this for the most interesting property, namely type safety, and illustrate the use of type information.

A program is type safe, if a terminating execution of a method or statement starting in a well–typed program state leads to a well–typed state. According to the explanation in section 3.4, a restricted version of this notion of type safety can be expressed in our logic by the following rule:

$$\frac{\vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\vdash \{\ \mathbf{P} \wedge \mathbf{TA}_{pre}\ \}\ \text{COMP}\ \{\ \mathbf{Q} \wedge \mathbf{TA}_{post}\ \}}$$

where the type annotations $\mathbf{TA}_{pre}$ and $\mathbf{TA}_{post}$ express that pre– and poststate are well–typed. These type annotations are made precise by the following definition:

**Definition 4.1 :** Adding Type Annotations
Let **A** be a triple; *adding type annotations* to **A** means to add type information about the parameters and variables to the pre- and postcondition of **A** and to require that the environment is well–typed. The resulting triple is denoted by *typed*(**A**). The type annotation of a triple **A** depends on its kind:

- If **A** is a method annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ meth T::m $\{\ \mathbf{Q}\ \}$, *typed*(**A**) is

  $$\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} typ(p_i) \preceq \text{TP}_i \wedge wt(\$)\ \}$$
  `T::m`
  $$\{\ \mathbf{Q} \wedge typ(\text{result}) \preceq \text{TR} \wedge wt(\$)\ \}$$

  where $p_i$, $i = 0, \ldots, y$, are the formal parameters of T::m with types $\text{TP}_i$ and TR denotes the result type of T::m.

- If **A** is a statement annotation, i.e. $\mathbf{A} \equiv \{\ \mathbf{P}\ \}$ STAT $\{\ \mathbf{Q}\ \}$, *typed*(**A**) is

  $$\{\ \mathbf{P} \wedge \bigwedge_{i=0}^{y} typ(p_i) \preceq \text{TP}_i \wedge \bigwedge_{i=0}^{z} typ(v_i) \preceq \text{TV}_i \wedge wt(\$)\ \}$$
  `STAT`
  $$\{\ \mathbf{Q} \wedge \bigwedge_{i=0}^{y} typ(p_i) \preceq \text{TP}_i \wedge \bigwedge_{i=0}^{z} typ(v_i) \preceq \text{TV}_i \wedge wt(\$)\ \}$$

  where $p_i$, $i = 0, \ldots, y$, are the formal parameters and $v_i$, $i = 0, \ldots, z$ are the local variables of the method enclosing STAT with types $\text{TP}_i$ and $\text{TV}_i$ respectively.

The *typed*–operator is canonically extended to sets $\mathcal{A}$ of triples.

□

**Lemma 4.2 :** Type Safety of AICKL*
If there exists a proof for $\mathcal{A} \vdash \mathbf{A}$, then $typed(\mathcal{A}) \vdash typed(\mathbf{A})$ can be proved.
□

**Proof**
The above lemma is proved by induction on the depth of the proof tree for $\mathcal{A} \vdash \mathbf{A}$. I.e. we have to look at all axioms and rules and show that they allow to derive the typed version of an annotation assuming that the annotation can be shown. Here we discuss only the most interesting cases.

**Induction Base:** A leaf of any proof tree is the instatiation of an axiom. For all axioms we have to show that if a triple is an instantiation of an axiom then the typed version of the triple can be proved as well. We demonstrate this for the most complex case, the write_att–axiom:

$$\begin{array}{l}
\{\ \mathrm{self} = S \wedge \mathrm{p} = P \wedge \$ = E \wedge \neg isvoid(S) \wedge typ(S) = \mathrm{T} \\
\quad \wedge\ typ(\mathrm{self}) \preceq \mathrm{T} \wedge typ(\mathrm{p}) \preceq rtyp(\mathrm{T} :: \mathrm{att}) \wedge wt(\$)\ \} \\
\Rightarrow \\
\{\ \mathrm{self} = S \wedge \mathrm{p} = P \wedge \$ = E \wedge \neg isvoid(S) \wedge typ(S) = \mathrm{T} \\
\quad \wedge \neg isvoid(S) \wedge typ(S) = \mathrm{T} \wedge typ(P) \preceq rtyp(\mathrm{T} :: \mathrm{att}) \wedge wt(E)\ \} \\
\hline
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \downarrow \quad [\![\ \text{inv–rule}\ ]\!] \\
\{\ \mathrm{self} = S \wedge \mathrm{p} = P \wedge \$ = E \wedge \neg isvoid(S) \wedge typ(S) = \mathrm{T}\ \} \\
\quad \texttt{meth T::write\_att( p )} \\
\{\ \$ = E\langle S.\mathrm{att} := P\rangle\ \} \\
\hline
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \uparrow \\
\{\ \$ = E\langle S.\mathrm{att} := P\rangle \\
\quad \wedge \neg isvoid(S) \wedge typ(S) = \mathrm{T} \wedge typ(P) \preceq rtyp(\mathrm{T} :: \mathrm{att}) \wedge wt(E)\ \} \\
\Rightarrow \quad [\![\ \text{see below}\ ]\!] \\
\{\ \$ = E\langle S.\mathrm{att} := P\rangle \wedge wt(\$)\ \}
\end{array}$$

The proof of the implication used above is identical to the proof for lemma 3.7 if we replace $typ(P) \preceq rtyp(\mathrm{T} :: \mathrm{att})$ for $typ(P) = rtyp(\mathrm{T} :: \mathrm{att})$.

**Induction Step:** Assume some instantiation of some proof rule is given. For the induction step, we have to derive the typed version of the succedent from the typed versions of the antecedents. Here we demonstrate it for the subtype rule, because this derivation shows where type constraints in subtyping come from, i.e. the contravariance of parameter typing and covariance of result typing. For the derivation we assume that the subtype rule is applied for a $\mathrm{T}' \preceq \mathrm{T}$, that $p_i$ and $p_i'$, $i = 1, \ldots, y$, are the explicit parameters of T::m and T′::m with types $\mathrm{TP}_i$ and $\mathrm{TP}_i'$ and that the result types are TR and TR′. The definition of subtyping (definition 2.1) implies $\mathrm{TP}_i \preceq \mathrm{TP}_i'$ and $\mathrm{TR}' \preceq \mathrm{TR}$. The derivation starts with the induction hypothesis, uses the type constraints for strengthening and weakening, and finally applies the subtype–rule:

$$\dfrac{\begin{array}{l}\bigwedge_{i=1}^{y} \mathit{typ}(p_i) \preceq \mathrm{TP}_i \;\Rightarrow\; \bigwedge_{i=1}^{y} \mathit{typ}(p_i) \preceq \mathrm{TP}'_i \\ \mathit{typed}(\mathcal{A}) \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}' \wedge \mathbf{P} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(p_i) \preceq \mathrm{TP}'_i \wedge \mathit{wt}(\$)\ \} \\ \quad \mathrm{meth}\ \mathrm{T}'{::}\mathrm{m}\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathrm{result}) \preceq \mathrm{TR}' \wedge \mathit{wt}(\$)\ \}\end{array}}{\begin{array}{l}\mathit{typed}(\mathcal{A}) \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}' \wedge \mathbf{P} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(p_i) \preceq \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \} \\ \quad \mathrm{meth}\ \mathrm{T}'{::}\mathrm{m}\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathrm{result}) \preceq \mathrm{TR}' \wedge \mathit{wt}(\$)\ \} \\ \mathit{typ}(\mathrm{result}) \preceq \mathrm{TR}' \;\Rightarrow\; \mathit{typ}(\mathrm{result}) \preceq \mathrm{TR}\end{array}}\ [\![\ \text{strengthening}\ ]\!]$$

$$\dfrac{}{\begin{array}{l}\mathit{typed}(\mathcal{A}) \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}' \wedge \mathbf{P} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(p_i) \preceq \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \} \\ \quad \mathrm{meth}\ \mathrm{T}'{::}\mathrm{m}\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathit{result}) \preceq \mathrm{TR} \wedge \mathit{wt}(\$)\ \} \\ \mathrm{T}' \preceq \mathrm{T}\end{array}}\ [\![\ \text{weakening}\ ]\!]$$

$$\dfrac{}{\begin{array}{l}\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T}' \wedge \mathbf{P} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(p_i) \preceq \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \} \\ \quad \mathrm{meth}\ \mathrm{T}{::}\mathrm{m}\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathit{result}) \preceq \mathrm{TR} \wedge \mathit{wt}(\$)\ \}\ ,\ \mathit{typed}(\mathcal{A}) \\ \quad \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T} \wedge \mathbf{P} \wedge \bigwedge_{i=1}^{y} \mathit{typ}(p_i) \preceq \mathrm{TP}_i \wedge \mathit{wt}(\$)\ \} \\ \quad\quad \mathrm{meth}\ \mathrm{T}{::}\mathrm{m}\ \{\ \mathbf{Q} \wedge \mathit{typ}(\mathit{result}) \preceq \mathrm{TR} \wedge \mathit{wt}(\$)\ \}\end{array}}\ [\![\ \text{subtype–rule}\ ]\!]$$

**end of proof**

**Using Type Information** The above lemma shows that AICKL* programs are type safe. As the type annotation is clear from the annotated component, we can and will assume in the following that *type annotations are implicitly present*, i.e. that each triple is typed even if the type information is not mentioned explicitly. As an example, we give the subtyping rule with type annotations kept implicit:

subtype–ta–rule:

$$\dfrac{\begin{array}{l}\mathrm{T}' \preceq \mathrm{T} \\ \mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{meth}\ \mathrm{T}'{::}\mathrm{m}\ \{\ \mathbf{Q}\ \}\end{array}}{\{\ \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T}' \wedge \mathbf{P}\ \}\ \mathrm{meth}\ \mathrm{T}{::}\mathrm{m}\ \{\ \mathbf{Q}\ \}\ ,\ \mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{meth}\ \mathrm{T}{::}\mathrm{m}\ \{\ \mathbf{Q}\ \}}$$

In the proof of the type safety lemma 4.2, the typed version of the subtype rule was derived from the untyped version. If we omit the first step in that derivation, we get the following stronger version with implicit typing:

$$\dfrac{\begin{array}{l}\mathrm{T}' \preceq \mathrm{T} \\ \{\ \mathbf{P} \wedge \bigwedge_{i=0}^{z} \mathit{typ}(\mathrm{p}_i) \preceq \mathrm{TP}_i\ \}\ \mathrm{meth}\ \mathrm{T}'{::}\mathrm{m}\ \{\ \mathbf{Q}\ \}\end{array}}{\{\ \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T}' \wedge \mathbf{P}\ \}\ \mathrm{meth}\ \mathrm{T}{::}\mathrm{m}\ \{\ \mathbf{Q}\ \} \vdash \{\ \mathbf{P}\ \}\ \mathrm{meth}\ \mathrm{T}{::}\mathrm{m}\ \{\ \mathbf{Q}\ \}}$$

This version of the subtype rule with implicit type annotations is stronger than the version above, because the proof of the (second) antecedent in this version may use stronger type constraints on the parameters, namely those type constraints corresponding to the parameter declaration for T::m (and not for T′::m).

As shown above, type information can be more precise than the static type information provided by parameter and variable declaration. The rest of this paragraph demonstrates the use of type information for program verification by a small example. Let us assume an abstract type T and variables v, w of type T. Let T1 be a concrete subtype of T, i.e. $\mathrm{T1} \preceq \mathrm{T}$, and let us assume that we have proved $\{\ \mathbf{P}\ \}\ \mathrm{T1::m}\ \{\ \mathbf{Q}\ \}$ for method m of interface T1. In the following statement, method T::m is called:

```
v := w . m()
```

We show how the dynamic type information $\mathit{typ}(w) \preceq \mathrm{T1}$ can be used to prove

$$\{\ \mathit{typ}(w) \preceq \mathrm{T1} \wedge \mathbf{P}[w/\mathrm{self}]\ \}\quad \mathrm{v := w\ .\ m()}\quad \{\ \mathbf{Q}[v/\mathrm{result}]\ \}$$

i.e. we do not need a full specification of T::m for all subtypes of T, but can use specifications of method m in subtypes of T, as long as we have enough type information about the self–object. From the subtype–ta–rule above we get:

$$\begin{array}{ll}
\mathrm{T1} \preceq \mathrm{T} & \\
\{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1} \wedge \mathbf{P}\ \}\ \mathrm{meth\ T1::m}\ \{\ \mathbf{Q}\ \} & \\
\hline
\{\ \mathit{typ}(\mathrm{self}) \not\preceq \mathrm{T1} \wedge \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1} \wedge \mathbf{P}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathbf{Q}\ \} & [\![\ \text{subtype–ta–rule}\ ]\!] \\
\qquad \vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1} \wedge \mathbf{P}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathbf{Q}\ \} & \\
\hline
\vdash \{\ \mathit{typ}(\mathrm{self}) \preceq \mathrm{T1} \wedge \mathbf{P}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathbf{Q}\ \} & [\![\ \text{false–axiom}\ ]\!] \\
\hline
\vdash \{\ \mathit{typ}(w) \preceq \mathrm{T1} \wedge \mathbf{P}[w/\mathrm{self}]\ \}\ \mathrm{v := w\ .\ m()}\ \{\ \mathbf{Q}[v/\mathrm{result}]\ \} & [\![\ \text{call–rule}\ ]\!]
\end{array}$$

## 4.2 Specifying Object–Oriented Programs

Specifications of object–oriented programs provide a formal means for program documentation and help to structure program proofs. Just as programs use imported interfaces (on the syntactic and implementation level), specified programs use the specifications of the imported interfaces. And program proofs are based on the proofs of the imported interface specifications. Consequently, the reuse of program modules would enable the "reuse of proofs". The comparison between program and proof reuse can illustrate an important point: Just as programs are only appropriate for reuse if they have a "good" and well–defined interface, program proofs are only appropriate for reuse if the proved program specifications are "good" and sufficiently expressive. In particular, it is not sufficient to specify only the functional behaviour of a program: For using program specifications in proofs, it is essential to know which properties remain invariant under program executions.

This section reviews the specification constructs from chapter 2, defines the meaning of these constructs in terms of triples, investigates the influence of program extensions on the correctness of specifications, and illustrates program specifications by several examples.

### 4.2.1 Specifying Program Properties

As introduced in chapter 2, an interface specification consists of a syntactic type interface, an interface invariant, a requires clause for each method, a set of pre- and postcondition pairs for each method, signatures of abstraction operations, and abstract data types to express the abstract behaviour of the interface. Pre- and postconditions for methods have to satisfy the context conditions explained in chapter 3. A requires clause has to satisfy the context conditions for preconditions. Interface invariants are specified by a $\Gamma$–formula[7] with (at most) one free variable being of sort *Object*. This section closes the gap between interface specifications as illustrated in chapter 2 and implementations as introduced in chapter 3. In particular, it demonstrates the specification of abstraction operations.

Considered as part of the software engineering process, the relation between interface specifications and implementations can be different:

1. Interface specifications can be an abstract documentation technique for given implementations.

2. Interface specifications can be design specifications as part of an entire software development framework.

3. Interface specifications and implementations can be developed together in an iterated and intertwined process.

As we are only interested in the formal relation between interface specifications and implementations, we are free to choose the point of view that is best for our needs. In this paragraph, we investigate interface specifications according to the first role. We assume that an implementation is given and develop an interface specification for it. As example, we use the interface and class PLIST. By the PLIST example, we can demonstrate several different aspects relevant for the specification and verification of interfaces. In particular, it illustrates the role of specifications for interfaces that are used for the implementation.

As already explained in section 2.3.2, PLIST is a subtype of LIST (i.e. an implementation of integer lists) allowing to iterate through lists and selectively update elements. PLIST is implemented by a class with three attributes. The first attribute references an array for storing the list elements. The second attribute records the size of the list, the third attribute records the current position of the iterator:

```
class  PLIST  is
   attr  arr: ARRAY
   attr  siz: INT
   attr  pos: INT
   ...
```

[7] Recall from the end of section 3.1.2 that $\Gamma$ is the signature that includes all program–independent symbols, the type and attribute constants, and the object environment constant \$.

```
      meth  append( n: INT ): PLIST  is
         ilv: PLIST; av: ARRAY; iv: INT;
         iv :=  self.siz ;
         iv :=  iv.add(1) ;
         av :=  self.arr ;
         av :=  av.resize(iv) ;
         av.set(iv,n) ;
         ilv     :=  new PLIST ;
         ilv.pos :=  iv ;
         ilv.arr :=  av ;
         ilv.siz :=  iv ;
         result :=   ilv
       end
       ...
   end
```

Method append is shown above because it is used in verification examples in section 4.3. The complete implementation of PLIST can be found in appendix A.6.

As the implementation of PLIST uses the implementation of ARRAY, the specification of PLIST will be based on the specification of ARRAY. To model the behaviour of interface ARRAY, we assume an abstract data type Array with functions $a\_size(A)$ yielding the size of an array $A$ and $a\_read(A, N)$ yielding the object held by $A$ at index $N$ (The complete signature of Array is given in section 4.2.4.) The abstraction functions for ARRAY–objects is denoted by $aA$.

In section 2.3.2, the data type PList is defined to model lists with positions: Lists with position are modeled by a pair of two lists such that the position is the first element of the second component. The function *pltol* is defined to embed sort *PList* into *List* by concatenating both list components. To define the well–formedness predicates and abstraction function for PLIST–objects, the auxiliary functions *arr*, *siz*, and *pos* are used that read the attribute values from a PLIST–object and take the corresponding abstraction:

$$\begin{array}{ll} arr & : \; Object \times ObjEnv \;\rightarrow\; Array \\ & typ(X) = ct(\text{PLIST}) \;\Rightarrow\; arr(X,E) = aA(E(X.\text{PLIST::arr}), E) \\ siz & : \; Object \times ObjEnv \;\rightarrow\; Integer \\ & typ(X) = ct(\text{PLIST}) \;\Rightarrow\; siz(X,E) = aI(E(X.\text{PLIST::siz})) \\ pos & : \; Object \times ObjEnv \;\rightarrow\; Integer \\ & typ(X) = ct(\text{PLIST}) \;\Rightarrow\; pos(X,E) = aI(E(X.\text{PLIST::pos})) \end{array}$$

Finally, we introduce a function that converts parts of an array into a list. As list elements in our example are always integers, whereas arrays may hold arbitrary objects, this conversion works only for arrays with integer objects. This restriction is checked by the predicate *intarray*. The conversion maps the array element with the highest index to the first list element:

$$\begin{array}{l} atol \; : \; Array \times Integer \times Integer \; \rightarrow \; List \\ \quad intarray(A) \wedge 0 \leq N \leq a\_size(A) \; \Rightarrow \; atol(A,N,N) = empt \\ \quad intarray(A) \wedge 0 \leq LN < UN \leq a\_size(A) \\ \qquad \Rightarrow \; atol(A,LN,UN) = app(aI(a\_read(A,UN)), atol(A,LN,UN-1)) \end{array}$$

$$\begin{array}{l} intarray : Array \; \rightarrow \; Boolean \\ \quad intarray(A) \Leftrightarrow_{def} \; \forall N : 0 < N \leq a\_size(A) \; \Rightarrow \; typ(a\_read(A,N)) = INT \end{array}$$

Based on these preliminaries, we first define the well–formedness predicate *wfP* for PLIST–objects and then the abstraction function *aP*:

$$\begin{array}{l} wfP \; : Object \times ObjEnv \; \rightarrow \; Boolean \\ \quad wfP(X,E) \Leftrightarrow_{def} \; typ(X) = ct(\mathrm{PLIST}) \wedge \neg isvoid(X) \wedge alive(X,E) \\ \qquad \wedge \; 0 \leq pos(X,E) \leq siz(X,E) \leq a\_size(arr(X,E)) \wedge intarray(arr(X,E)) \end{array}$$

$$\begin{array}{l} aP \; : Object \times ObjEnv \; \rightarrow \; PList \\ \quad wfP(X,E) \; \Rightarrow \; aP(X,E) = plist( \; atol(arr(X,E), pos(X,E), siz(X,E)), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad atol(arr(X,E), 0, pos(X,E)) \; ) \end{array}$$

As the abstraction function is used to explain the functional behaviour of the methods of PLIST, the well–formedness predicate has to be satisfied by the parameters and results of the methods. Thus, the well–formedness predicate is a natural candidate for an invariant. For a reason that is explained below the length of lists represented by PLIST–objects is bound by *maxint*:

```
interface  PLIST  subtype of LIST  is
   inv X: wfP(X,$) /\ lng(pltol(aP(X,$))) <= maxint
   ...
```

As example for method specifications, we consider specifications for the methods append and set. The specification of the method append consists of the requires clause and two pre–post–pairs:

```
meth  append( n: INT ): PLIST
   req   lng(pltol(aP(self,$))) < maxint

   pre   pltol(aP(self,$)) = L /\ n! = N
   post  aP(result,$) = plist( empt, app(N,L) )

   pre   E<<$
   post  E<<$
```

The requires clause enforces the length of the list represented by a self–object to be less than *maxint*. Without this requirement, the length of the resulting list could not be held by attribute *siz*. (This requirement is a typical example for conditions that turn up during verification.) The first pre–post–pair specifies the functional behaviour of append: If $L$ denotes the list represented by the self–object and $N$ denotes the integer to be appended, the result is the list $app(N, L)$ where the position is reset to the first element. The second pre–post–pair specifies that append produces no side–effects, i.e. it does not change locations of objects living in the prestate (cf. section 3.2).

A more complex example is method set. The method set requires that the self–object has a well–defined position (i.e. the back–component of the abstraction is not empty). The method set changes the element at the current position to the given parameter:

```
   meth  set( e: INT )
      req   ~isempt(back(aP(self,$)))

      pre   self = S /\ e! = E /\
            aP(S,$) = plist( FL, app(EB,RL) )
      post  aP(S,$) = plist( FL, app(E, RL) )
  end
```

The method set produces a side–effect. The problem for a user of the interface PLIST is to know which objects are affected by calls to method set and how this can be proved. From the specification above, we can only deduce that objects being disjoint from the self–object are not affected (cf. lemma 3.12). But e.g. from the specification of append, we can not deduce whether the object produced by a call to append is disjoint from the object upon which append was called. Properties of this kind will be called *sharing properties* and are further investigated after the meaning of interface specification is clarified and further specification techniques are developed.

### 4.2.2 The Meaning of Interface Specifications

The program related part of interface specifications consists of requires clauses, pre–post–pairs, and invariants (cf. section 2.2.2). In this subsection, we will further analyze these three constructs and give them a precise meaning in terms of triples. Thus, we consider an interface specification as an abbreviation for a set of triples. Consequently, verifying an interface specification reduces to prove the corresponding set of triples.

#### Invariants

We start our analysis of interface specification constructs with invariants. Essentially, we have to answer three questions:

- What does it mean that an invariant holds in a state?
- In which execution states should an invariant hold or what is the role of invariants in verification?
- How can the meaning of invariants be expressed in terms of pre- and postconditions?

An invariant of a declared interface T can be considered as a predicate $inv_{\mathrm{T}}(X,E)$ with one parameter of sort *Object* and one parameter of sort *ObjEnv*. This predicate is defined by the invariant clause for T: The invariant clause $INV(X)$ associated with interface T is a Γ–formula with at most one free variable $X$. The formula $INV(X)$ describes the properties that should hold for any living non–void object of type T in some of the execution states. As Γ includes the global variable \$ for the object environment, invariants may depend on the state. These considerations are summarized in the definition of the invariant predicate:

$$inv_{\mathrm{T}}(X,E) \Leftrightarrow_{def} \; typ(X) \preceq \mathrm{T} \wedge \neg isvoid(X) \wedge alive(X,E) \;\Rightarrow\; INV(X)[E/\$] \quad (4.7)$$

The formula $\forall X : inv_{\mathrm{T}}(X,\$)$ is called the *closed form* of the invariant or just the *closed invariant*. Certainly, an invariant need not hold in all states. E.g. the invariant of the interface PLIST will not hold in intermediate states during the execution of method PLIST::empty :

```
(1)   meth  empty(): PLIST  is
(2)      ilv: PLIST; av: ARRAY; iv: INT;
(3)      av      :=  ARRAY::create(0) ;
(4)      ilv     :=  new PLIST ;
(5)      ilv.arr :=  av ;
(6)      ilv.pos :=  0 ;
(7)      ilv.siz :=  0 ;
(8)      result  :=  ilv
(9)   end
```

In execution states between line (4) and (5), the newly created PLIST–object does not satisfy the well–formedness predicate for PLIST which is a conjunct of the invariant for PLIST.

The above example suggests that an invariant for interface T should hold in all states that are "outside" of an execution of a T–method. In order to make this meaning of invariants more precise, we have to eliminate illegal executions. Interface properties can only be guaranteed if methods are used correctly, i.e. if the requires clause is satisfied in the prestate of a method execution. We define an execution to be *legal*, if it only consists of correct method executions. An invariant for interface T has to hold in all states of legal executions that are not part of an execution of a T–method. It suffices to require this property for all public T–methods, because private methods of interface T are only executed inside public methods.

The explanation of invariants given above is based on an informal operational semantics of AICKL* (cf. [LW94], chapter 3). We use this explanation here as a guideline. For verification purposes, it has two fairly big disadvantages:

- It is too complex to formalize it in a Hoare–style logic.
- It does not make any statement about the properties of T–objects within T–methods. Thus, we may not assume invariant properties at calls to T–methods that occur within T–methods.

The meaning of invariants in terms of triples, i.e. a meaning that is compatible with Hoare–style verification, is given in the following definition.

**Definition 4.3 :** Meaning of Invariants
Let $\mathrm{T}_1, \ldots, \mathrm{T}_p$ be the interfaces of a module M with closed invariants $INV_{\mathrm{T}_i}$. We say that $INV_{\mathrm{T}_1}, \ldots, INV_{\mathrm{T}_p}$ are invariants of a module M if their conjunction $INV_{\mathrm{M}}$ remains invariant for each method of a public interface, i.e. if the following triples can be proved for each method m of a public interface T with requires clause $REQ(\mathrm{T::m})$

$$\{\ REQ(\mathrm{T::m}) \wedge INV_{\mathrm{M}}\ \}\ \mathrm{meth\ T::m}\ \{\ INV_{\mathrm{M}}\ \} \tag{4.8}$$

□

The rational for this definition is as follows: $INV_{\mathrm{M}}$ holds in the initial state of a program execution when only integer, boolean and void objects exist, because invariants are only concerned with properties of living non–void objects. The triples guarantee that $INV_{\mathrm{M}}$ is invariant for all public methods. The requirement that the *conjunction* of all invariants have to be invariant for all public methods might seem unnecessarily complicate. In subsection 4.2.3.2, we analyze the reasons for this.

### Method Requirements

In order to work properly, a method requires that certain properties hold in the prestate. These requirements may come from the fact that the implemented function is partial (e.g. method rest requires that the self–object is non–empty), or result from implementation constraints as illustrated by the requirements for the append method above. The requirements have to describe as well the correct data representation of the arguments, e.g. that singly linked lists are not cyclic. Whereas partiality and implementation constraints are method specific, data representation depends on the implementation of the whole interface (e.g. the well–formedness condition for PLIST). In an interface specification, method specific requirements are stated in the requires clauses. Aspects of data representation are usually formulated as invariants. Thus, the meaning of a requires clause can be expressed as follows:

$$\{\ REQ(\mathrm{T::m}) \wedge INV_{\mathrm{M}}\ \}\ \mathrm{meth\ T::m}\ \{\ \mathrm{TRUE}\ \} \tag{4.9}$$

As these formulas can be derived from the invariant formulas above, they need not be part of the proof obligations resulting from an interface specification.

### Pre–Post–Pairs

Beside invariants and requires clauses, an interface specification provides for each method m a set of pre–post–pairs. Using a set of pre–post–pairs instead of just one pair, as it is done in many interface specification languages (in particular in Larch interface languages: cf. [GH93]), makes method specifications more readable: The specification of the method append above gave a first example illustrating that method specifications describe different aspects, e.g. the functional behaviour and the absence of side–effects (cf. 2.2.1.1).

After the discussion in the previous paragraph, the meaning of pre–post–pairs in the context of interface specifications should be clear. Beside the precondition of the pre–post–pair, the requires clause and the invariants can be assumed to hold in the prestate. Thus, a pre–post–pair (**P**,**Q**) for method T::m abbreviates the following triple:

$$\{\ \mathit{REQ}(\text{T::m}) \wedge \mathit{INV}_{\text{M}} \wedge \mathbf{P}\ \}\ \text{meth T::m}\ \{\ \mathbf{Q}\ \} \tag{4.10}$$

Altogether the meaning of an interface specification is given by a set of triples: One triple for each requires clause and one triple for each pre–post–pair of the public methods.

### 4.2.3 Discussing Interface Specification Constructs

The previous section explained the meaning of interface specifications as introduced in chapter 2 in terms of Hoare triples. What are the advantages of using interface specification constructs compared to a direct use of Hoare triples? We advise the use of interface specification constructs because of two reasons: 1. They lead to shorter specifications. 2. They provide structure. If triples were used directly, invariants and method requirements have to be explicitly put into the preconditions for all methods. This would inflate the preconditions and would make modifications to invariant and requirement aspects error prone, because all occurrences would have to be modified consistently.

Structuring an interface specification into invariants, method requirements, and pre–post–pairs has advantages beyond better readability and modifiability. We will illustrate two aspects here: 1. Rearranging pre–post–pairs; 2. Analysis of program extensions. The importance of structuring specifications for mechanical specification and verification support is briefly discussed in chapter 5.

#### 4.2.3.1 Rearranging Pre–Post–Pairs

Rearranging pre–post–pairs means to shift parts of the precondition into the post-condition and vice versa. In general, rearranging is only possible in the context of requirements that guarantee the correct execution of the considered component. Thus, for rearranging method annotations the requires clause is needed. We illustrate rearranging by an example and formulate a lemma about rearranging.

As an example, we consider a pre–post–pair of the array specification presented in section 4.2.4:

$$\{\ \mathbf{R} \wedge T \neq \text{ARRAY} \wedge X = \mathit{new}(\$, T)\ \}\ \texttt{ARRAY} :: \texttt{create}\ \{\ X = \mathit{new}(\$, T)\ \}$$

where **R** denotes the method requirement for create. Now, we like to derive from this annotation that method create only creates objects of type ARRAY, i.e.:

$$\{\ \mathbf{R} \wedge \neg \mathit{alive}(X, \$)\ \}\ \texttt{ARRAY} :: \texttt{create}\ \{\ \mathit{alive}(X, \$)\ \Rightarrow\ \mathit{typ}(X) = \text{ARRAY}\ \}$$

The derivation would be a simple consequence[8] of axiom env12 if we had given both annotations in the following form:

$$\{\ \mathbf{R} \wedge E = \$\ \}\ \texttt{create}\ \{\ T \neq \text{ARRAY} \wedge X = \mathit{new}(E,T)\ \Rightarrow\ X = \mathit{new}(\$,T)\ \}$$

$$\{\ \mathbf{R} \wedge E = \$\ \}\ \texttt{create}\ \{\ \neg\mathit{alive}(X,E)\ \Rightarrow\ (\mathit{alive}(X,\$) \Rightarrow \mathit{typ}(X) = \text{ARRAY})\ \}$$

The difference between both forms is that parts of the preconditions are shifted into the postcondition by introducing a logical variable for the prestate–environment. The advantage of this later form is that free variables occurring in pre– and postcondition can be universally quantified. E.g.:

$$\{\ \mathbf{R} \wedge E = \$\ \}$$

$$\overline{\hspace{20em}}\downarrow \quad [\![\ \text{all–rule}\ ]\!]$$

$$\{\ \mathbf{R} \wedge E = \$\ \}$$

```
ARRAY::create
```

$$\{\ T \neq \text{ARRAY} \wedge X = \mathit{new}(E,T)\ \Rightarrow\ X = \mathit{new}(\$,T)\ \}$$

$$\overline{\hspace{20em}}\uparrow$$

$$\{\ \forall X : T \neq \text{ARRAY} \wedge X = \mathit{new}(E,T)\ \Rightarrow\ X = \mathit{new}(\$,T)\ \}$$

$$\Rightarrow$$

$$\{\ T \neq \text{ARRAY}\ \Rightarrow\ \mathit{new}(E,T) = \mathit{new}(\$,T)\ \}$$

On the other hand, the form given first is often better to read and more appropriate for proving aspects about method calls.

Rearranging of pre–post–pairs is important for interface specification and verification: If rearranging was not possible without proving the rearranged pair from scratch, pre–post–pairs would have to be designed with a lot of care in order to find the most suitable form. Fortunately, rearranging in both directions is possible for any component COMP, in particular for methods, if we know the requirements **R** guaranteeing the legal execution of COMP, i.e. $\{\ \mathbf{R}\ \}$ COMP $\{$ TRUE $\}$. As rearranging is a common operation during verification, we formalize the concept in the following lemma:

**Lemma 4.4 :** Rearranging
Let **P** be a $\Sigma$–formula and COMP be a component having the program variables[9] $v_1, \ldots, v_k$ in its context. Furthermore let **R** and **Q** be pre– and postformulas for COMP such that $\{\ \mathbf{R}\ \}$ COMP $\{$ TRUE $\}$. Then the following two triples are equivalent, i.e. given one triple we can prove the other:

$$\{\ \mathbf{R} \wedge \mathbf{P}[v_1/V_1, \ldots, v_k/V_k]\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}$$

$$\{\ \mathbf{R} \wedge \bigwedge_{i=1}^{k} v_i = V_i\ \}\ \text{COMP}\ \{\ \mathbf{P}\ \Rightarrow\ \mathbf{Q}\ \}$$

$\square$

[8]The first postcondition gives $\mathit{new}(E,T) = \mathit{new}(\$,T)$ for all $T \neq$ ARRAY; for any $X$ with $\neg\mathit{alive}(X,E) \wedge \mathit{alive}(X,\$)$ axiom env12 yields $\mathit{typ}(X) \neq T$ for all $T \neq$ ARRAY, i.e. $\mathit{typ}(X) =$ ARRAY .

[9]Recall that \$ and parameters are considered as program variables as well.

The proof of the lemma above is straightforward.

#### 4.2.3.2 Invariants and Program Extensions

Having invariants as an explicit concept of interface specifications simplifies the analysis of programs and program extensions. In particular, invariants isolate those properties of imported interfaces that cause new proof obligations in extended programs. Slightly simplifying, the following proof obligations arise: 1. The invariants of the imported interfaces have to hold for the declared methods. 2. The invariants of the declared interfaces have to hold for the imported methods. Both kinds of proof obligations result from the requirement that the conjunction of all interface invariants has to remain invariant under all public methods of a program.

This subsection illustrates where such seemingly discouraging load of proof obligations comes from and studies techniques to improve the situation for practical applications. In addition to demonstrating the role of invariants in interface specifications, the verification aspects discussed here serve as motivation and preparation for interface specification techniques presented in the next section.

**Invariance Under All Methods** As a first step towards a better understanding of the proof obligations sketched above, we study their source using the following module REF_MOD. Interface REFOBJ provides the functionality of an object reference: The reference can be created, dereferenced, and updated. Interface CREFREFINT uses REFOBJ to implement constant references to references to integers:

```
module  REF_MOD  import STD_MOD  is

   concrete interface  REFOBJ  subtype of  OBJECT  is
      include  OBJECT
      meth  create(): REFOBJ
      meth  deref (): OBJECT
      meth  update( o: OBJECT )
   end

   concrete interface  CREFREFINT  subtype of  OBJECT  is
      include  OBJECT
      meth  create(): CREFREFINT
      meth  deref (): REFOBJ
   end

   class  REFOBJ  is
      attr  cont: OBJECT
      meth  create(): REFOBJ  is
         result := new REFOBJ;
         result. update( 0 )
      end
      meth  deref (): OBJECT  is
         result := self.cont
      end
      meth  update( o: OBJECT )  is
         self.cont := o
      end
   end

   class  CREFREFINT  is
      attr  ref: REFOBJ
      meth  create(): CREFREFINT  is
         rov: REFOBJ;
         rov    :=  REFOBJ::create();
         result :=  new CREFREFINT;
         result. ref := rov
      end
      meth  deref (): REFOBJ  is
         result := self.ref
      end
   end
   public  REFOBJ, CREFREFINT, OBJECT, INT, BOOL
end
```

According to our definition invariants have to hold for all public methods. It is clear that the invariant of an interface T has to hold for the methods of T. Why must it hold for all other public methods as well? To demonstrate this, we consider the following interface specification for CREFREFINT:

```
concrete interface  CREFREFINT  subtype of  OBJECT  is
   inv X:  wellformed(X,$)
   include  OBJECT
   meth  create(): CREFREFINT
      req  TRUE
   meth  deref (): REFOBJ
      req  ~isvoid(self)
end
```

where the well–formedness predicate guarantees that referenced objects are non–void objects referencing an integer:

$$\begin{aligned}&\mathit{wellformed}(X,E) \Leftrightarrow_{def}\\ &\quad \neg \mathit{isvoid}(X) \wedge \neg \mathit{isvoid}(E(X.\mathit{ref})) \wedge \mathit{typ}(E(E(X.\mathit{ref}).\mathit{cont})) = \mathit{INT}\end{aligned}$$

The invariant for CREFREFINT holds for all methods of CREFREFINT, but may be violated by method `REFOBJ::update`, e.g. in the following fragment:

```
crriv :=  CREFREFINT::create();
rov   :=  crriv . deref() ;
rov . update( true )
```

At the end of this fragment, variable crriv holds an object that violates the invariant: The indirectly referenced object is not an integer. That is why we have to prove the invariants for all public methods. In our example, this cannot be done, because method `update` may produce side–effects violating the invariant for CREFREFINT.

**Constraining Invariants** Invalidation of invariants as illustrated by the example above comes from object sharing and selective updating. Another source of problems with invariants is the generality of invariant specifications. Whereas problems of the first kind cannot be solved in general[10], problems of the second kind can be avoided. We briefly illustrate problems of the second kind and then show how to avoid them by semantical constraints on invariants.

An invariant of interface T should express properties of T–objects and objects that are referenced by T–objects (e.g. the invariant of PLIST formulates properties of PLIST–objects and referenced ARRAY–objects). But, an invariant of interface T can as well express properties of other objects. For instance, we could add the following formula as a conjunct to the invariant of interface CREFREFINT:

$$\forall Z : \mathit{alive}(Z, \$) \wedge \mathit{typ}(Z) \neq \mathrm{CREFREFINT} \wedge \mathit{typ}(Z) \neq \mathrm{REFOBJ} \ \Rightarrow\ \mathit{static}(Z)$$

This addendum requires that no objects of types different from CREFREFINT and REFOBJ are created. It expresses a property, namely non–aliveness, of objects that can not be referenced from CREFREFINT– or REFOBJ–objects. It is an invariant for all methods of CREFREFINT and REFOBJ, but any sensible program extending CREFREFINT and REFOBJ would violate the invariant. This is certainly an undesirable situation. Unfortunately, it is fairly complex, if not impractical to design syntactic restrictions of invariants that exclude such situations without overly reducing the expressiveness of invariants. As alternative, we use a *logical constraint*: If an invariant $\mathit{inv}_{\mathrm{T}}$ holds for $X$ in an environment $E$ and $E'$ is X–equivalent to $E$, then the invariant has to hold as well in $E'$ (and vice versa):

$$E \equiv_X E' \ \Rightarrow\ (\mathit{inv}_{\mathrm{T}}(X, E) \ \Leftrightarrow\ \mathit{inv}_{\mathrm{T}}(X, E')) \tag{4.11}$$

The above restriction creates a proof obligation for each invariant. For example for the extended invariant of CREFREFINT given above does not satisfy this *logical constraint*. Thus, it is not admissible as invariant.

**Proof Obligations from Program Extension** With the above discussion in mind, we can look at proof obligations about invariants resulting from program extensions. We denote the conjunction of the imported interfaces by $INV_{imp}$ and the conjunction of the declared interfaces by $INV_{dcl}$. We have to show that $INV_{imp} \wedge INV_{dcl}$ is invariant for all imported and all declared methods. For most practical applications, this is equivalent to the following four kinds of proof obligations:

1. $INV_{imp}$ is invariant for all imported methods.
2. $INV_{imp}$ is invariant for all declared methods.

[10]At least not as long as we want to support the efficiency of selective updates.

3. $INV_{dcl}$ is invariant for all imported methods.

4. $INV_{dcl}$ is invariant for all declared methods.

Obligations of kind 1 are part of the verification of the imported interfaces. Obligations of kind 2 are studied below (cf. in particular lemma 4.7). For obligation of kind 3, there are essentially two cases according to the relation between a declared and an imported interface: Either the declared interface uses the imported interface for its implementation (like for example PLIST uses ARRAY) or it does not. In the first case, the imported interface should be declared private, because its methods may in general violate the representation of the declared and implemented interface if used in an inappropriate way (e.g. method ARRAY::set may violate the invariant of a PLIST–representation). As invariants must hold only for public methods, nothing has to be shown. This case is further discussed in the following subsection. In the second case, showing the invariance of $INV_{dcl}$ for the methods of an imported unrelated interface is typically a simple task if the interface specification of the imported interfaces is sufficiently expressive. Obligations of kind 4 constitute the actual part of the proof and can of course not be reduced. Techniques for proving this kind of obligations are explained and illustrated in section 4.3.

Like obligations of kind 1, obligations of kind 2 might seem to hold as well in all program extensions as long as invariants satisfy 4.11. Unfortunately, subtyping can introduce dependencies between imported and declared interfaces making it possible that $INV_{imp}$ is violated by declared methods. We show this by a tiny example. Then, we formulate sufficient conditions under which obligations of kind 2 may be discarded without extra proof.
As example, we consider the following extension of module REF_MOD from above:

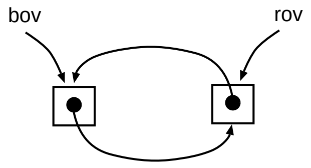
```
module  BAD_MOD  import REF_MOD  is
   concrete interface  BADOBJ  subtype of OBJECT  is
      include  OBJECT
      meth  new(): BADOBJ
      meth  write_badref( o: OBJECT ): BADOBJ
   end
   class  BADOBJ  is
      attr badref:  OBJECT
   end
   public  BADOBJ, REFOBJ, CREFREFINT, OBJECT, INT, BOOL
end
```

As specification of the imported interface REFOBJ we assume:

```
concrete interface  REFOBJ  subtype of  OBJECT  is
   inv X: ~reach($(X.cont),X.cont,$)
   include  OBJECT
   meth  create(): REFOBJ   req  TRUE
   meth  deref (): OBJECT   req  ~isvoid(self)
   meth  update(o: OBJECT)  req  ~isvoid(self) /\ disj(self,o,$)
end
```

The invariant clause states that a REFOBJ–object never reaches itself; i.e. chains of objects are acyclic. In the program having REF_MOD as top module, this invariant can be guaranteed, because chains of linked objects consist only of REFOBJ–objects and because the only method to update these objects requires that the self–object is disjoint from the explicit parameter. But in the extended program with module BAD_MOD, the invariant for REFOBJ does not longer hold as demonstrated by the following program fragment:

```
bov :=  new BADOBJ;
rov :=  REFOBJ::create();
rov . update( bov );
bov.badref := rov
```



Method write_badref can violate the invariant for REFOBJ by introducing a cycle via a BADOBJ–object; i.e. the invariant $INV_{imp}$ of the imported interface(s) does not hold for a declared method. The invariant of REFOBJ is violated because of the following two reasons:

1. REFOBJ has an attribute of type OBJECT and the declared interface is (of course) a subtype of OBJECT;

2. the invariant imposes restrictions on objects held by attribute cont; these restrictions are guaranteed by the imported methods, but not by the declared methods.

In more general terms, the first reason says that the implementation of imported interfaces has an attribute with range type T and one of the declared types is a subtype of T. In the following, we show that the absence of such subtype relations guarantees that invariants of imported interfaces hold for declared methods. Another possibility to avoid proving obligations of kind 2 is to check whether the second reason is given.

To formulate subtype relations as sketched above, we need the following definitions:

**Definition 4.5 :** Reachability of Types
$treach_n(T, N, T')$ denotes the fact that type $T$ reaches type $T'$ by stepping N–times from a type to the range type of one of its attributes; more precisely:

$$treach_n(T, 0, T') \;\Leftrightarrow\; T \succeq T'$$

$$treach_n(T, N+1, T') \;\Leftrightarrow\; \exists A : treach_n(T, N, dtyp(A)) \wedge rtyp(A) \succeq T'$$

The fact that a type $T$ reaches a type $T'$ is denoted by $treach(T, T')$ and defined as:

$$treach(T, T') \;\Leftrightarrow\; \exists N : treach_n(T, N, T')$$

□

Reachability of types is a static approximation of reachability of objects:

**Lemma 4.6 :** Approximating Reachability
In a well–typed environment, a location is reachable from an object $X$ only if its domain type is reached by the type of $X$; more precisely:

$$typ(X) \preceq T \wedge typ(obj(L)) = T' \wedge \neg treach(T, T') \;\Rightarrow\; \neg reach(X, L, E)$$

□

Proof by induction on the number of reaching steps.

**Lemma 4.7 :** Invariance of Imported Invariants
Let M be a module declaring the interfaces $\mathrm{TD}_1, \ldots, \mathrm{TD}_k$ and importing the interfaces $\mathrm{TI}_1, \ldots, \mathrm{TI}_l$, OBJECT; let $inv_{imp}(X, E)$ denote the conjunction of all invariant predicates for interfaces $\mathrm{TI}_1, \ldots, \mathrm{TI}_l$ and $INV_{imp}$ the corresponding closed form. If no imported type $\mathrm{TI}_i$ reaches a declared type[11], i.e. $\neg treach(\mathrm{TI}_i, \mathrm{TD}_j)$, then $INV_{imp}$ is invariant under all declared methods dm; more generally:
If we can prove $\mathcal{A} \vdash \{\ \mathbf{P}\ \}$ COMP $\{\ \mathbf{Q}\ \}$ for a program component of M using the interface specifications of imported methods as axioms,
then we can prove $invadd(\mathcal{A}) \vdash \{\ \mathbf{P} \wedge INV_{imp}\ \}$ COMP $\{\ \mathbf{Q} \wedge INV_{imp}\ \}$
where $invadd(\mathcal{A})$ is obtained from $\mathcal{A}$ by adding $INV_{imp}$ to the pre– and postcondition of the assumptions.

□

**Proof**
The above lemma is proved by induction on the depth of the proof tree for $\mathcal{A} \vdash \{\ \mathbf{P}\ \}$ COMP $\{\ \mathbf{Q}\ \}$ (for the proof technique cf. the proof of the type safety lemma 4.2).

**Induction Base:** The interface specifications of imported methods already have $INV_{imp}$ as conjunct in pre– and postconditions (cf. section 4.2.2). How $INV_{imp}$ is added to the pre– and postconditions of the assumpt– and false–axiom was shown in the proof of lemma 3.7. Adding $INV_{imp}$ to the pre– and postconditions of axioms about methods can be straightforwardly proved correct for cases where the current object environment is not changed. Remains to show it for the write–att–axiom and the new–axiom:

- write–att–axiom: According to the first rule derived in section 3.3 (see page 59), it suffices to show the following triple:

  $\{\ \mathrm{self}{=}S \wedge \mathrm{p}{=}P \wedge \${=}E \wedge \neg isvoid(S) \wedge typ(S) = \mathrm{TD} \wedge inv_{imp}(X, E)\ \}$
  meth TD::write_att( p)
  $\{\ \$ = E\langle S.\mathrm{att} := P\rangle \wedge inv_{imp}(X, E\langle S.\mathrm{att} := P\rangle)\ \}$

  where TD is a concrete type in $\{\mathrm{TD}_1, \ldots, \mathrm{TD}_k\}$. This is proved by case distinction on the free variable $X$:

[11]This is in particular satisfied, if declared types are not subtypes of the imported types $\mathrm{TI}_i$.

Case 1: $typ(X) \not\preceq \mathrm{TI}_i$ for all $\mathrm{TI}_i$. Thus, $inv_{\mathrm{TI}_i}(X, E)$ holds for all $\mathrm{TI}_i$ and arbitrary $E$ (cf. definition of $inv_{\mathrm{T}}$ in section 4.2.2). And as $inv_{imp}(X, E)$ is the conjunction of the $inv_{\mathrm{TI}_i}(X, E)$, it holds as well for arbitrary $E$; in particular, we get

$$inv_{imp}(X, E) \;\Leftrightarrow\; inv_{imp}(X, E\langle S.\mathrm{att} := P\rangle) \tag{4.12}$$

enabling us to prove the triple by the inv–rule.

Case 2: $typ(X) \preceq \mathrm{TI}_i$ for one $\mathrm{TI}_i$. In this case, we derive the equivalence 4.12 from the general constraint 4.11 on invariants. To establish $E \equiv_X E\langle S.\mathrm{att} := P\rangle$ we have to show

*i)* $alive(X, E) \;\Leftrightarrow\; alive(X, E\langle S.\mathrm{att} := P\rangle)$

*ii)* $reach(X, L, E) \;\Rightarrow\; E(L) = E\langle S.\mathrm{att} := P\rangle)(L)$

The first obligation is a consequence of env6–axiom. The second obligation follows from the env1–axiom if $L \neq S.\mathrm{att}$. Otherwise lemma 4.6 together with $typ(X) \preceq \mathrm{TI}_i$ and $typ(obj(S.\mathrm{att})) = typ(S) = \mathrm{TD}$ and $\neg treach(\mathrm{TI}_i, \mathrm{TD})$ yields $\neg reach(X, L, E)$.

- new–axiom: Essentially, we have to show:

$$inv_{imp}(X, E) \;\Leftrightarrow\; inv_{imp}(X, E\langle \mathrm{TD}\rangle)$$

If $X \neq new(E, \mathrm{TD})$ this follows directly from lemma 3.4.(iv) and 4.11. Otherwise we have $typ(X) = \mathrm{TD}$. According to the first case above it suffices to show $typ(X) \not\preceq \mathrm{TI}_i$ for all $\mathrm{TI}_i$; i.e. $\mathrm{TD} \not\preceq \mathrm{TI}_i$ for all $\mathrm{TI}_i$. This is a direct consequence of $\neg treach(\mathrm{TI}_i, \mathrm{TD})$.

**Induction Step:** As $INV_{imp}$ does not contain local variables or parameters, the cases for the induction steps are straightforward.

**end of proof**

#### 4.2.3.3 Interface Specifications and Encapsulation

Interface specifications and encapsulation facilities are closer related than it might be understood from the explanations above. Meaningful invariants only exist if the programming language allows to hide methods. Otherwise objects could be created and arbitrarily initialized by public methods. Thereby any invariant that restricts the possible states of objects could be invalidated. E.g. in a PLIST–object the size–attribute could be modified violating the well–formedness condition in the invariant (cf. section 4.2.1).

Declaring entire interfaces to be private (as it is demonstrated for interface ARRAY in subsection 4.1.1.2) allows to encapsulate methods and type names. A hidden type name cannot be used as range type of attributes in program extensions. Thus, it can be guaranteed that in all program extensions objects of private types can only be referenced through objects of public types. In section 4.3.2, we demonstrate how such information can be used for verification.

### 4.2.4 Revisiting Interface Specifications

Section 4.2.1 illustrated the central constructs of interface specifications and showed how interface specifications are linked to the formal data and state model. In this section, we look at interface specifications from a methodological point of view. Beside providing the abstract data types to model interface behaviour, interface specifications are concerned with three different aspects:

1. They describe the *functional* behaviour of methods, i.e. they specify the (functional) relation between the abstraction of input parameters and the result parameter of methods.

2. They describe *environmental* behaviour of methods, i.e. they specify what parts of the object environment are modified and which properties of the environment remain invariant.

3. They can describe the *sharing* behaviour of methods, i.e. specify properties about representations of abstract values by linked objects.

Designing "good" interface specifications is not a simple task. An important quality criterion for interface specifications is e.g. whether they are sufficiently expressive to verify modules that use the interface. Together with the three specification aspects, the following paragraphs provide a first analysis of the relation between a) abstract data types and interface specifications, b) different method annotations, and c) interface specifications and implementations.

#### 4.2.4.1 Specifying Functional Behavior

The standard technique to specify the functional behaviour of an interface was already illustrated in section 2.2: 1. Define an abstraction function that maps objects in an environment to values of an abstract data type. 2. Specify for each method of the interface how the abstraction of the method's result is expressed in terms of the abstractions of the parameters. In addition to this, the requires clauses that guarantee legal execution have to be specified. Based on this technique, the design of such specifications mainly consists of choosing an appropriate abstract data type. Many different abstraction levels are possible. The abstract data type can in particular be based on data type Object and can abstract only from parts of the representations by linked objects. We demonstrate this by specifying the functional behaviour of interface ARRAY (cf. section 4.1.1.2) based on the abstract data type Array with the following signature:

$$\begin{array}{lcll}
a\_init & : & Integer & \rightarrow & Array \\
a\_write & : & Array \times Integer \times Object & \rightarrow & Array \\
a\_read & : & Array \times Integer & \rightarrow & Object \\
a\_size & : & Array & \rightarrow & Integer \\
a\_resize & : & Array \times Integer & \rightarrow & Array
\end{array}$$

$$\begin{array}{ll}
0 < N \leq a\_size(A) & \Rightarrow a\_read(a\_write(A,N,X),N) = X \\
0 < N \leq a\_size(A) \wedge N \neq N' & \Rightarrow a\_read(a\_write(A,N',X),N) = a\_read(A,N) \\
0 < N \leq min(a\_size(A), SIZE) & \Rightarrow a\_read(a\_resize(A,SIZE),N) = a\_read(A,N)
\end{array}$$

$$\begin{array}{llll}
0 < N & \Rightarrow & a\_size(a\_init(N)) & = N \\
0 < N & \Rightarrow & a\_size(a\_write(A,N,X)) & = a\_size(A) \\
0 < N & \Rightarrow & a\_size(a\_resize(A,N)) & = N
\end{array}$$

$$aA: \quad Object \times ObjEnv \rightarrow Array$$

For positive arguments $N$, function *a_init* yields an array of size $N$ with lower bound 1 and upper bound $N$; $a\_write(A,N,X)$ sets the array $A$ at index $N$ to object $X$, if $N$ is within the bounds of $A$; $a\_read(A,N)$ reads $A$ at index $N$; *a_size* yields the size of an array, and $a\_resize(A,N)$ yields an array of size $N$ that is in the first $K$–elements identical to $A$ where $K$ is the minimum of $N$ and $size(A)$. The function $aA$ abstracts objects of type ARRAY in an environment to values of sort *Array*.

The interface ARRAY provides the basic operations to handle arrays, i.e. to create an array, to get the element of an array at a given index, to set the array component at a given index to a given element, and to resize an array. Based on the above abstract data type, the requirements for and the functional behaviour of the ARRAY–methods are specified as follows:

```
meth  size(): INT
   post  result! = a_size( aA(self,$)^ )

meth  get( n: INT ): OBJECT
   req   0 < n! <= a_size(aA(self,$))
   post  result = a_read( aA(self,$)^, n!^ )

meth  set( n: INT, x: OBJECT )
   req   0 < n! <= a_size(aA(self,$))
   post  aA(self^,$) = a_write(aA(self,$)^,n!^,x^)

meth  create( size: INT ): ARRAY
   req   0 <= size!
   post  a_size(aA(result,$)) = size!^  /\
         ( 0 < N <= size!^  => a_read(aA(result,$),N) = int(0) )

meth  resize( size: INT ): ARRAY
   req   0 <= size!
   post  aA(result,$) = a_resize(aA(self,$)^,size!^)
```

Method size provides the most simple example to illustrate the relation between object level and abstract level. Method get illustrates a typical requires clause. Method set shows how methods without results can be treated. Method create demonstrates that the correspondence between methods and functions on the abstract level need not be one–to–one. In particular, the result of a method can be specified by a formula in a loose way. I.e. in general the specification of the functional method behaviour only expresses some relational properties between the method parameters and the method result. This looseness is desirable in particular for the specification of abstract interfaces.

#### 4.2.4.2 Specifying Environmental Behaviour

The object environment can be considered as an in–out–parameter of each method. From a formal point of view, environmental behaviour is specified just as functional or sharing behaviour. Two aspects give specifications of environmental behaviour a special role: 1. Environment operations are language–defined. Thus, a program–independent specification technique can be developed. 2. Environment specifications affect all classes and all objects. If an environment specification of an interface T is too weak, it may disable us to prove invariant properties about objects of types T′ that are not related to T. In this subsection, we show how language–defined operations are used to specify environmental behaviour and discuss techniques to meet the problems related to the second aspect.

To demonstrate environmental specifications, we use again the interface ARRAY concentrating on the methods get, set, and create (size is treated similar to get and resize similar to create). The simplest case is if a method does not modify the environment:

```
meth  get( n: INT ): OBJECT
   post  $^ = $
```

A method does not produce side–effects if it does not modify locations of objects being alive in the prestate. In section 3.2 the predicate $\ll$ on environments is introduced capturing this relation. Method create and method resize do not produce side–effects:

```
meth  create( size: INT ): ARRAY
   post  $^ << $

meth  resize( size: INT ): ARRAY
   post  $^ << $
```

This environmental specification gives no hints about objects created by the method. The simplest way to overcome this problem is to guarantee explicitly that method create (and similar method resize) only allocates objects of type ARRAY (cf. subsection 4.2.3.1). We call this the *local creation property*:

```
meth  create( size: INT ): ARRAY
   post  T # ARRAY =>  new($^,T) = new($,T)
```

Finally, we have to specify the environmental behaviour of method set. Method set guarantees that the environment remains equivalent for all objects that do not reach the self–parameter:

```
meth  set( n: INT, x: OBJECT )
   pre   ~oreach(Y,self,$)
   post  $^ ≡_Y $

   post  new($^,T) = new($,T)
```

The second pre–post–pair simply states that method set does not create objects. Based on the given environmental specification, we like to discuss how quality criterions for such interface specifications can be developed. A more systematic analysis of this problem is considered as a topic for future research (cf. chapter 5). The basic idea to analyze the applicability of specifications is to ask questions and check whether such questions can be answered by the specification: The question is a conjecture, the specification can answer iff the conjecture can be proved. We demonstrate this technique by a positive and negative example.

**Alien Types** In general, methods of an interface T can modify all objects that are reachable from their parameters by calling methods for these objects. Through subtyping, they can even modify objects of types that are declared in program extensions. To simplify verification, it is important to know for which objects environments remain equivalent under method execution. I.e. if T::m is a method of interface T, we look for a predicate $p(X, \mathrm{T})$ such that

$$\{\ \mathbf{R} \wedge p(X,\mathrm{T}) \wedge E = \$\ \}\quad \mathrm{T::m}\quad \{\ E \equiv_X \$\ \}$$

where **R** stands for the requirement and invariant of T::m. Formulated in terms of types, we need a predicate $pt(S, \mathrm{T})$ such that equivalence of $E$ and \$ are guaranteed for all objects of type $S$. If a type $S$ reaches a type $T$, modifications to objects of type $T$ can influence objects of type $S$. Thus, a first candidate for $pt(S,T)$ is $\neg treach(S,T)$. This candidate has to be refined for cases where $T$ is an abstract type, because $S$ should not reach subtypes of $T$:

**Definition 4.8 :** Alien Types
We call a type $S$ alien to $T$ if $S$ does not reach a subtype of $T$:

$$alien(S,T) \;\Leftrightarrow\; \forall T' : T' \preceq T \;\Rightarrow\; \neg treach(S,T')$$

□

If type $S$ is alien to type $T$, creation or modification of $T$–objects are not visible from $S$–objects:

**Lemma 4.9 :** Alien Properties

i) $alien(S,T) \wedge ct(T') \preceq T \;\Rightarrow\; E \equiv_S E\langle T' \rangle$

ii) $alien(S,T) \wedge typ(obj(L)) \preceq T \;\Rightarrow\; E \equiv_S E\langle L := Y \rangle$

□

The proof of this lemma is an application of lemma 4.6.

**Testing Environmental Specifications** Based on the preliminary considerations about alien types, the question to the environmental specification of interface ARRAY is formulated as follows: Can we prove

$$\{\ \mathbf{R} \wedge \mathit{alien}(T, \mathrm{ARRAY}) \wedge E = \$\ \}\quad \mathrm{ARRAY::m}\ \ \{\ E \equiv_T \$\ \}$$

for all methods of interface ARRAY from the above interface specification? This property in particular guarantees that methods of interface ARRAY do not modify objects reachable from an array (at least as long as these objects do not reach arrays). Methods size and get do not modify the environment, i.e. keep the environment equivalent for all objects. Methods create and resize do not create objects of alien type. Thus, objects of alien type are alive in the prestate iff they are alive in the poststate, and by rearranging lemma 4.4, we can prove the desired property. The interesting case is method set:

$\{\ \mathbf{R} \wedge \mathit{typ}(\mathrm{self}) = \mathrm{ARRAY} \wedge \mathit{alien}(T, \mathrm{ARRAY}) \wedge E = \$\ \}$

$\overline{\hspace{12em}}\downarrow\ \ [\![\ \text{rearranging and all–rule}\ ]\!]$

$\{\ \mathbf{R} \wedge \mathit{typ}(\mathrm{self}) = \mathrm{ARRAY} \wedge \mathit{alien}(T, \mathrm{ARRAY}) \wedge \mathit{typ}(Y) \preceq T \wedge E = \$\ \}$

$\Rightarrow\ \ [\![\ \text{simple derivation using lemma 4.6}\ ]\!]$

$\{\ \mathbf{R} \wedge \neg \mathit{oreach}(Y, \mathrm{self}, \$) \wedge E = \$\ \}$

```
meth set( n: INT, x: OBJECT )
```

$\{\ E \equiv_Y \$\ \}$

$\overline{\hspace{12em}}\uparrow$

$\{\ \forall Y : \mathit{typ}(Y) \preceq T\ \Rightarrow\ E \equiv_Y \$\ \}$

$\Rightarrow$

$\{\ E \equiv_T \$\ \}$

The additional conjunct $\mathit{typ}(\mathrm{self}) = \mathrm{ARRAY}$ is contained in the type annotations that are always assumed (cf. paragraph “Using Type Information” in section 4.1.3). Thus, we can prove the conjecture for all methods of interface ARRAY.

To provide an example for a question that cannot be answered by the environmental specification of interface ARRAY, we consider the following finer conjecture:

$\{\ \mathbf{R} \wedge \mathit{extern}(T, \mathrm{ARRAY\_MOD}) \wedge \mathit{typ}(\mathit{obj}(L)) \preceq T \wedge \$(L) = X\ \}$

```
ARRAY::m
```

$\{\ \$(L) = X\ \}$

I.e. locations of objects that are of a type extern to module ARRAY_MOD are not modified by methods of interface ARRAY. This property cannot be derived from the given environmental specifications. The specification of method set is too weak as it does not state properties of objects that reach the self–parameter. The property could be derived from an implementation of ARRAY[12].

[12] For the special case of module ARRAY, the property could have been derived as well based on syntactical module properties, i.e. if we had provided a formalization of the module concept (cf. section 4.1.2).

As an interface specification is based on abstractions, one cannot expect that it is complete in the logical sense, i.e. that it allows to prove all properties of the implementation. One of the key issues of interface specifications is to hide implementation details. This information hiding concerns in particular environmental and sharing properties. Maybe future research reveals hard criterions for comprehensive interface specifications. Up to now we use a kind of check list mechanism: 1. Specify under which conditions a method maintains environment equivalence. 2. If a method creates objects, specify for which types no objects are created. 3. Establish disjointness or non–reachability properties where possible. The above interface specification of ARRAY violates the last point. At least for method create we would expect that its result cannot be reached by objects being alive in the prestate:

```
meth  create( size: INT ): ARRAY
   pre   alive(Y,$)
   post  ~oreach(Y,result,$)
```

In the following we assume this property as well for method resize. But, resize methods in arrays do not necessarily have this property; i.e. guaranteeing this property for method resize is more a design question than a question of providing a comprehensive environmental specification of the interface ARRAY. This example shows again how close interface design and the comprehensive specification of interfaces are.

#### 4.2.4.3 Specifying Sharing Properties

Subsection 2.2.1.1 introduced the specification of sharing behaviour. It discussed a sharing property of objects of type LIST. Sharing may range over small data structures like in the LIST example or over very big data structures, e.g. if two objects share a data base. This paragraph investigates the relation between sharing properties and implementations using the list example. It first reviews the abstract sharing properties, then analyzes the method annotations for interface LIST, and finally relates it to the list implementations.

Many list implementations support sharing of representations; i.e. two lists may use common objects for their representation. Without knowing the concrete represenation of lists, we can specify sharing properties. Similar to abstract data type specification, functions and predicates expressing the sharing behaviour are specified by their properties. These properties are formulated by first–order axioms as well as by triples. Some of these properties are invariant properties, others are properties expressed by method annotations. For the list example, sharing properties are expressed by two predicates: *dirpart* states that a list *XL* is a direct part of a list *YL*; *part* states that a list *XL* is part of a list *YL*. Predicates *dirpart*, *part*, and the auxilary predicate $part_n$ have the following signature and basic properties:

$$dirpart \;:\; Object \times Object \times ObjEnv \;\rightarrow\; Boolean$$

$$\begin{array}{ll} \mathit{part} & : \mathit{Object} \times \mathit{Object} \times \mathit{ObjEnv} \;\rightarrow\; \mathit{Boolean} \\ \mathit{part}_n & : \mathit{Object} \times \mathit{Object} \times \mathit{Nat} \times \mathit{ObjEnv} \;\rightarrow\; \mathit{Boolean} \end{array}$$

$$\begin{array}{l} \mathit{part}_n(XL, YL, 0, E) \quad\;\; \Leftrightarrow\; \mathit{wfL}(XL, E) \wedge XL = YL \\ \mathit{part}_n(XL, YL, N+1, E) \;\Leftrightarrow\; \mathit{wfL}(XL, E) \wedge \mathit{wfL}(YL, E) \\ \qquad\qquad \wedge\; \exists ZL : \mathit{dirpart}(XL, ZL, E) \wedge \mathit{part}_n(ZL, YL, N, E) \end{array}$$

$$\begin{array}{l} \mathit{part}(XL, YL, E) \;\Leftrightarrow\; \exists N : \mathit{part}_n(XL, YL, N, E) \\ \mathit{dirpart}(XL, YL, E) \;\Rightarrow\; \mathit{wfL}(XL, E) \wedge \mathit{wfL}(YL, E) \wedge \mathit{aL}(XL, E) = \mathit{rst}(\mathit{aL}(YL, E)) \end{array}$$

where *wfL* expresses well–formedness of lists (we assume in particular $\mathit{wfL}(XL, E) \Rightarrow \mathit{alive}(XL, E)$). The first three axioms specify the fact that *part* is the reflexive, transitive closure of *dirpart*. The last axiom establishes a relation between *dirpart* and the abstraction function *aL* for lists. Based on these axioms we can prove e.g. that

$$\mathit{dirpart}(XL, YL, E) \;\Rightarrow\; \neg\mathit{part}(YL, XL, E) \tag{4.13}$$

As invariant sharing properties, we specify that two well–formed lists are either disjoint or share a common part:

$$\begin{array}{l} \forall XL, YL : \;\; \mathit{wfL}(XL, \$) \wedge \mathit{wfL}(YL, \$) \\ \qquad \Rightarrow\;\; \mathit{disj}(XL, YL, \$) \vee \exists ZL : \mathit{part}(ZL, XL, \$) \wedge \mathit{part}(ZL, YL, \$) \end{array}$$

Specifying a property as an invariant is a weaker statement than specifying it as an axiom. Axioms have to hold for all environments, invariants have to hold only in states that result from method executions where the invariant property is assumed for the prestates. E.g. doubly linked list implementations (cf. figure 2.1) usually do not satisfy the above property in arbitrary environments — the back part of one list can be the front part of the other —, but their methods often maintain this property.

The central properties of *dirpart* and *part* are specified as method annotations. As the LIST–methods empty, isempty, and first do not modify the environment, they maintain in particular the sharing properties. As the sharing predicate implies liveness of objects, the no–side–effect–property of method rest and append enable to derive that they maintain the sharing property too. For method updfst, we have to specify it explicitly:

```
meth  updfst( n: INT )  is
   pre   dirpart(XL,YL,$)
   post  dirpart(XL,YL,$)
```

This pre–post–pair allows in particular to derive that method updfst maintains as well the part–relation.

In order to use sharing properties in verification, there has to be at least one method annotation that establishes a sharing property in the postcondition without assuming it in the precondition. In our example, the direct–part–relation is established by method rest:

```
meth  rest(): LIST  is
   post  dirpart(result,self^,$)
```

The main application of sharing properties is to improve the precision and generality of functional and environmental specifications. E.g. based on predicate *part*, we can give a very accurate account of the behaviour of method updfst:

```
meth  updfst( n: INT )  is
   pre   ~part(self,XL,$)
   post  aL(XL,$)^ = aL(XL,$)

   pre   part(self,XL,$) /\ aL(XL,$)=conc(PREFIX,aL(self,$)))
   post  aL(XL,$) = conc( PREFIX, app( n^!, rst( aL(self,$))^ )) )
```

**Relating Sharing Specifications to Implementations** The above axioms, the invariant property, and the method annotations specify sharing properties in an implementation–independent way. To underline this fact, we have provided the specification for the abstract interface LIST that as two different implementations (cf. subsection 4.1.1.2): the array–based implementation PLIST (cf. section 4.2.1) and a singly–linked list implementation CLIST sketched in section 3.1.1.

Beside the class and method declarations, implementations have to provide definitions for the abstraction, well–formedness, and sharing functions. Based on these definitions, the interface properties can be proved. To illustrate these aspects, we have a closer look at the list example. The definition of abstraction functions and well–formedness predicates was already demonstrated for PLIST (cf. section 4.2.1). We assume appropriate definitions of the abstraction function $aC$ and the well–formedness predicate $wfC$ for CLIST. The functions $aP$, $aC$, $wfP$, $wfC$ provide the basis to define the corresponding functions for abstract type LIST. To focus on the essential aspects, we assume here that type LIST is closed, i.e. CLIST and PLIST are its only subtypes (cf. section 4.1.2). Under this assumption the definitions of $wfL$ and $aL$ are:

$$\begin{array}{ll} wfL & : Object \times ObjEnv \ \rightarrow \ Boolean \\ & wfL(X,E) \ \Leftrightarrow \ (typ(X) = ct(\text{PLIST}) \wedge wfP(X,E)) \\ & \qquad\qquad\qquad \vee \ (typ(X) = ct(\text{CLIST}) \wedge wfC(X,E)) \end{array}$$

$$\begin{array}{ll} aL & : Object \times ObjEnv \ \rightarrow \ List \\ & typ(X) \preceq ct(\text{PLIST}) \ \Rightarrow \ aL(X,E) = pltol(aP(X,E)) \\ & typ(X) \preceq ct(\text{CLIST}) \ \Rightarrow \ aL(X,E) = aC(X,E) \end{array}$$

Finally, we define the sharing predicate *dirpart*:

$$\begin{array}{l} dirpart(X,Y,E) \ \Leftrightarrow \\ \quad (\ wfP(X,E) \wedge wfP(Y,E) \wedge E(X.\text{arr}) = E(Y.\text{arr}) \wedge E(X.\text{siz}) + 1 = E(Y.\text{siz})\ ) \\ \vee\ (\ wfC(X,E) \wedge wfC(Y,E) \wedge aC(Y,E) \neq empt \wedge X = E(Y.\text{tail})\ ) \end{array}$$

Based on such definitions, sharing properties as those illustrated above can be formally proved.

## 4.3 Verifying Object-Oriented Programs

In this section, we demonstrate the application of the developed programming logic and study essential aspects for proving object-oriented programs correct. The first part illustrates the use of interface specifications for the verification of programs. The second part explains techniques for the verification of invariants. Verifying invariants is different from verifying other properties, because invariants quantify over all living objects of a type and this quantification range changes when new objects are created.

### 4.3.1 Using Interface Specifications for Verification

Program verification is usually concerned with the verification of properties of complete programs. Program specification can help to structure these proofs: By considering a program as an extension of verified program parts, the specifications of the program parts can be used for the verification of the whole program. In this section, we discuss three aspects. First, we demonstrate how a specification of an abstract interface can be used to verify a method specification. Then, we discuss the verification of an abstract interface based on the interface specifications of the subtypes. Finally, we illustrate the role of invariants and abstraction functions in more detail.

#### A Simple Proof Using an Abstract Interface Specification

Section 2.2 explained the interface specification for interface LIST. Here, we use this interface specification to prove that the method sort given in section 4.1.1.2, p. 79, really sorts lists of type LIST. As LIST is an abstract interface, method sort actually works on objects of type CLIST and of type PLIST. But this fact is irrelevant for the proof, because it is encapsulated in the proof for the interface specification of LIST.

The sorting property is formulated using a function *a_sort* that sorts elements of sort *List*. With the help of *a_sort*, the specification of the method sort is straightforward:

```
meth  sort( l: LIST ): LIST
   pre   aL(l,$) = L
   post  aL(result,$) = a_sort(L)
```

To prove this specification, we need the corresponding property for the auxiliary method sorted_ins:

```
meth  sorted_ins( n: INT; l: LIST ): LIST
   pre   aL(l,$) = a_sort(L) /\ n! = N /\ lng(aL(l,$)) < maxint
   post  aL(result,$) = a_sort(app(N,L))
```

As method sorted_ins performs the comparison between integers, the proof for method sort only needs the following properties of *a_sort*:

$$lng(a\_sort(L)) = lng(L)$$
$$a\_sort(empt) = empt$$

The proof outline for the specification of method sort is given in figure 4.2. In the proof outline, the invariant properties for lists could be kept implicit, because all environment updates are done by methods for which the invariants hold. (The use of class invariants is illustrated in the following subsection.) The proof for sorted_ins is contained in appendix A.7. Although these two proofs are about methods that handle linked object structures, they are relatively simple, because the aspects concerned with data representations are encapsulated in the interface specification for LIST.

### Proving Functional Properties

This subsection demonstrates how abstract properties of implementations are verified. It illustrates the use of class invariants and the need of the X–equivalence property of abstraction functions. It shows where and how the different definitions that are part of an interface specification are used. As an example, we prove the functional specification of PLIST::append; i.e.:

$$\{\ \mathit{lng}(\mathit{pltol}(\mathit{aP}(\mathrm{self},\$))) < \mathit{maxint} \wedge \mathit{INV}$$
$$\wedge\ \mathit{pltol}(\mathit{aP}(\mathrm{self},\$)) = \mathit{SELF} \wedge \mathrm{n!} = N\ \}$$

```
meth append( n: INT ): PLIST
```

$$\{\ \mathit{aP}(\mathrm{result},\$) = \mathit{plist}(\mathit{empt}, \mathit{app}(N, \mathit{SELF}))\ \}$$

The proof is presented in some detail in order to show that such proofs are not complex, but technically difficult to handle without machine support. We develop the proof starting from the postcondition. Application of the write– and new–axiom and the introduction of the abbreviations $E$ and $W$ yield:

$$\{\ \exists E, W :\ \mathit{aP}(W,E) = \mathit{plist}(\mathit{empt}, \mathit{app}(N, \mathit{SELF})) \wedge W = \mathit{new}(\$, \mathrm{PLIST})$$
$$\wedge\ E = \$\langle \mathrm{PLIST}; W.\mathrm{pos} := \mathrm{iv}; W.\mathrm{arr} := \mathrm{av}; W.\mathrm{siz} := \mathrm{iv}\rangle\ \}$$

```
ilv := new PLIST ;
ilv.pos := iv ;
ilv.arr := av ;
ilv.siz := iv ;
```

$$\{\ \mathit{aP}(\mathrm{ilv},\$) = \mathit{plist}(\mathit{empt}, \mathit{app}(N, \mathit{SELF}))\ \}$$

```
result := ilv
```

$$\{\ \mathit{aP}(\mathrm{result},\$) = \mathit{plist}(\mathit{empt}, \mathit{app}(N, \mathit{SELF}))\ \}$$

The above precondition refers to different environments, namely $E$ and \$. A typical intermediate step in a program proof is to eliminate occurrences of \$ that are followed by new–operations or updates; i.e. to eliminate $E$ in the above precondition. For the example, we have to look for a property of `av`, `iv`, and \$ implying $\mathit{aP}(W,E) = \mathit{plist}(\mathit{empt}, \mathit{app}(N, \mathit{SELF}))$. It suffices to show that array av represents the correct list in environment \$, that av is an array of INT–objects and that the variable iv is a

```
    $\{\ aL(\mathrm{l},\$) = L\ \}$
bv := l.isempty()
    $\{\ \mathrm{bv!} = \mathit{isempt}(L) \wedge aL(\mathrm{l},\$) = L\ \}$
if bv then
    $\{\ \mathrm{bv!} \wedge \mathrm{bv!} = \mathit{isempt}(L) \wedge aL(\mathrm{l},\$) = L\ \}$
    $\Rightarrow$
    $\{\ L = \mathit{empt} \wedge aL(\mathrm{l},\$) = \mathit{empt}\ \}$
    $\Rightarrow\ \llbracket\ \mathit{a\_sort}(\mathit{empt}) = \mathit{empt}\ \rrbracket$
    $\{\ aL(\mathrm{l},\$) = \mathit{a\_sort}(L)\ \}$
result := l
    $\{\ aL(\mathrm{result},\$) = \mathit{a\_sort}(L)\ \}$
else
    $\{\ \neg\mathrm{bv!} \wedge \mathrm{bv!} = \mathit{isempt}(L) \wedge aL(\mathrm{l},\$) = L\ \}$
    $\Rightarrow$
    $\{\ \neg\mathit{isempt}(aL(\mathrm{l},\$)) \wedge aL(\mathrm{l},\$) = L \wedge \neg\mathit{isempt}(L)\ \}$
lv := l.rest(); $\llbracket$ inv. of $aL(\mathrm{l},\$) = L$ from no side–effects of rest $\rrbracket$
    $\{\ aL(\mathrm{lv},\$) = \mathit{rst}(L) \wedge aL(\mathrm{l},\$) = L \wedge \neg\mathit{isempt}(L)\ \}$
    $\Rightarrow\ \llbracket$ the (implicit) invariant yields $\mathit{lng}(aL(\mathrm{l},\$)) \le \mathit{maxint}\ \rrbracket$
    $\{\ \neg\mathit{isempt}(aL(\mathrm{l},\$)) \wedge aL(\mathrm{lv},\$) = \mathit{rst}(L) \wedge aL(\mathrm{l},\$) = L$
      $\wedge\ \neg\mathit{isempt}(L) \wedge \mathit{lng}(\mathit{rst}(L)) < \mathit{maxint}\ \}$
nv := l.first(); $\llbracket$ first does not change $\$$ $\rrbracket$
    $\{\ \mathrm{nv!} = \mathit{fst}(L) \wedge aL(\mathrm{lv},\$) = \mathit{rst}(L) \wedge \neg\mathit{isempt}(L) \wedge \mathit{lng}(\mathit{rst}(L)) < \mathit{maxint}\ \}$
    $\Rightarrow$
    $\{\ aL(\mathrm{lv},\$) = \mathit{rst}(L) \wedge \mathrm{nv!} = \mathit{fst}(L) \wedge \mathit{lng}(\mathit{rst}(L)) < \mathit{maxint} \wedge \neg\mathit{isempt}(L)\ \}$
lv := sort( lv );
    $\{\ aL(\mathrm{lv},\$) = \mathit{a\_sort}(\mathit{rst}(L)) \wedge \mathrm{nv!} = \mathit{fst}(L)$
      $\wedge\ \mathit{lng}(\mathit{rst}(L)) < \mathit{maxint} \wedge \neg\mathit{isempt}(L)\ \}$
    $\Rightarrow$
    $\{\ aL(\mathrm{lv},\$) = \mathit{a\_sort}(\mathit{rst}(L)) \wedge \mathrm{nv!} = \mathit{fst}(L)$
      $\wedge\ \mathit{lng}(aL(\mathrm{lv},\$)) < \mathit{maxint} \wedge \neg\mathit{isempt}(L)\ \}$
result := sorted_ins(nv, lv)
    $\{\ aL(\mathrm{result},\$) = \mathit{a\_sort}(\mathit{app}(\mathit{fst}(L), \mathit{rst}(L)) \wedge \neg\mathit{isempt}(L)\ \}$
    $\Rightarrow$
    $\{\ aL(\mathrm{result},\$) = \mathit{a\_sort}(L)\ \}$
end
    $\{\ aL(\mathrm{result},\$) = \mathit{a\_sort}(L)\ \}$
end
```

Figure 4.2: Proof outline for method sort

legal index for that array; i.e. we have to show the following implication:

$$\begin{array}{ll} & \mathit{atol}(\mathit{aA}(\mathrm{av},\$),0,\mathrm{iv!}) = \mathit{app}(N,\mathit{SELF}) \\ & \;\wedge\; \mathit{intarray}(\mathit{aA}(\mathrm{av},\$)) \wedge 0 \le \mathrm{iv!} \le \mathit{a\_size}(\mathit{aA}(\mathrm{av},\$)) \\ \Rightarrow & \\ & \exists E, W : \mathit{aP}(W,E) = \mathit{plist}(\mathit{empt},\mathit{app}(N,\mathit{SELF})) \wedge W = \mathit{new}(\$,\mathrm{PLIST}) \\ & \quad \wedge\; E = \$\langle \mathrm{PLIST}; W.\mathrm{pos} := \mathrm{iv}; W.\mathrm{arr} := \mathrm{av}; W.\mathrm{siz} := \mathrm{iv} \rangle \end{array}$$

To proof this implication, let $W$ and $E$ be defined as in the conclusion. The proof consists of four steps: 1. $E$ is *av*–equivalent to \$ because only locations of a new object are updated (cf. lemma 3.4.(iii)). Use the fact that the abstraction function *aP* yields the same result in equivalent environments. 2. Use the first property of *atol*. 3. $W$ is of type PLIST; use the abbreviations $\mathit{arr}(W,E)$, $\mathit{pos}(W,E)$, and $\mathit{siz}(W,E)$ (cf. section 4.2.1); according to env1– and env2–axiom, they equal $\mathit{aA}(\mathrm{av},E)$, iv!, and iv!. 4. Apply the definitions of *wfP* and *aP*:

$$\begin{array}{ll} & \mathit{atol}(\mathit{aA}(\mathrm{av},\$),0,\mathrm{iv!}) = \mathit{app}(N,\mathit{SELF}) \\ & \;\wedge\; \mathit{intarray}(\mathit{aA}(\mathrm{av},\$)) \wedge 0 \le \mathrm{iv!} \le \mathit{a\_size}(\mathit{aA}(\mathrm{av},\$)) \\ \Rightarrow & \\ & \mathit{atol}(\mathit{aA}(\mathrm{av},E),0,\mathrm{iv!}) = \mathit{app}(N,\mathit{SELF}) \\ & \;\wedge\; \mathit{intarray}(\mathit{aA}(\mathrm{av},E)) \wedge 0 \le \mathrm{iv!} \le \mathit{a\_size}(\mathit{aA}(\mathrm{av},E)) \\ \Rightarrow & \\ & \mathit{atol}(\mathit{aA}(\mathrm{av},E),0,\mathrm{iv!}) = \mathit{app}(N,\mathit{SELF}) \\ & \;\wedge\; \mathit{atol}(\mathit{aA}(\mathrm{av},E),\mathrm{iv!},\mathrm{iv!}) = \mathit{empt} \\ & \;\wedge\; \mathit{intarray}(\mathit{aA}(\mathrm{av},E)) \wedge 0 \le \mathrm{iv!} \le \mathrm{iv!} \le \mathit{a\_size}(\mathit{aA}(\mathrm{av},E)) \\ \Rightarrow & \\ & \mathit{atol}(\mathit{arr}(W,E),0,\mathit{pos}(W,E)) = \mathit{app}(N,\mathit{SELF}) \\ & \;\wedge\; \mathit{atol}(\mathit{arr}(W,E)),\mathit{pos}(W,E),\mathit{siz}(W,E)) = \mathit{empt} \\ & \;\wedge\; \mathit{intarray}(\mathit{arr}(W,E)) \wedge 0 \le \mathit{pos}(W,E) \le \mathit{siz}(W,E) \le \mathit{a\_size}(\mathit{arr}(W,E)) \\ \Rightarrow & \\ & \mathit{aP}(W,E) = \mathit{plist}(\mathit{empt},\mathit{app}(N,\mathit{SELF})) \end{array}$$

To continue the proof of PLIST::append we have to establish the premise of the above implication as a postcondition of a call to ARRAY::set (cf. section 4.2.4). Adapting a method specification to a given postcondition is done by rearranging. For our example we get after simplification:

```
meth  set( ix: INT, x: OBJECT )
   req   0 < ix! <= a_size(aA(self,$))
   pre   self = S /\ Z = a_write(aA(self,$),ix!,x)
   post  aA(S,$) = Z
```

Using this specification and a similar rearrangement of the specification of method resize, the postconditions can be pushed upwards through the program and that is the direction the following proof outline should be read:

$$\{\ \$(\text{self.siz})! + 1 \leq \mathit{maxint}$$
$$\wedge\ Y = \mathit{a\_resize}(\mathit{aA}(\$(\text{self.arr}), \$), \$(\text{self.siz})! + 1)$$
$$\wedge\ Z = \mathit{a\_write}(Y, \$(\text{self.siz})! + 1, \text{n})$$
$$\wedge\ \mathit{atol}(Z, 0, \$(\text{self.siz})! + 1) = \mathit{app}(N, \mathit{SELF})$$
$$\wedge\ \mathit{intarray}(Z) \wedge 0 \leq \$(\text{self.siz})! + 1 \leq \mathit{a\_size}(Z)\ \}$$

```
iv := self.siz ;
iv := iv.add(1) ;
av := self.arr ;
```

$$\{\ Y = \mathit{a\_resize}(\mathit{aA}(\text{av}, \$), \text{iv}!) \wedge Z = \mathit{a\_write}(Y, \text{iv}!, \text{n})$$
$$\wedge\ \mathit{atol}(Z, 0, \text{iv}!) = \mathit{app}(N, \mathit{SELF})$$
$$\wedge\ \mathit{intarray}(Z) \wedge 0 \leq \text{iv}! \leq \mathit{a\_size}(Z)\ \}$$

```
av := av.resize(iv) ;
```

$$\{\ \mathit{aA}(\text{av}, \$) = Y \wedge Z = \mathit{a\_write}(Y, \text{iv}!, \text{n})$$
$$\wedge\ \mathit{atol}(Z, 0, \text{iv}!) = \mathit{app}(N, \mathit{SELF})$$
$$\wedge\ \mathit{intarray}(Z) \wedge 0 \leq \text{iv}! \leq \mathit{a\_size}(Z)\ \}$$
$$\Rightarrow$$
$$\{\ Z = \mathit{a\_write}(\mathit{aA}(\text{av}, \$), \text{iv}!, \text{n})$$
$$\wedge\ \mathit{atol}(Z, 0, \text{iv}!) = \mathit{app}(N, \mathit{SELF})$$
$$\wedge\ \mathit{intarray}(Z) \wedge 0 \leq \text{iv}! \leq \mathit{a\_size}(Z)\ \}$$
$$\Rightarrow\ [\![\ \mathit{a\_size}(\mathit{a\_write}(A, N, E)) = \mathit{a\_size}(A)\ ]\!]$$
$$\{\ 0 < \text{iv}! \leq \mathit{a\_size}(\mathit{aA}(\text{av}, \$))$$
$$\wedge\ Z = \mathit{a\_write}(\mathit{aA}(\text{av}, \$), \text{iv}!, \text{n}) \wedge \mathit{atol}(Z, 0, \text{iv}!) = \mathit{app}(N, \mathit{SELF})$$
$$\wedge\ \mathit{intarray}(Z) \wedge 0 \leq \text{iv}! \leq \mathit{a\_size}(Z)\ \}$$

```
av.set(iv,n) ;
```

$$\{\ \mathit{aA}(\text{av}, \$) = Z \wedge \mathit{atol}(Z, 0, \text{iv}!) = \mathit{app}(N, \mathit{SELF})$$
$$\wedge\ \mathit{intarray}(Z) \wedge 0 \leq \text{iv}! \leq \mathit{a\_size}(Z)\ \}$$

It remains to be shown that the precondition of PLIST::append implies the precondition derived so far; i.e. we have to prove:

$$\mathit{pltol}(\mathit{aP}(\text{self}, \$)) = \mathit{SELF} \wedge \text{n}! = N \wedge \mathit{INV} \wedge \mathit{typ}(\text{self}) = \text{PLIST}$$
$$\Rightarrow$$
$$\exists Y, Z :\quad Y = \mathit{a\_resize}(\mathit{aA}(\$(\text{self.arr}), \$), \$(\text{self.siz})! + 1)$$
$$\wedge\ Z = \mathit{a\_write}(Y, \$(\text{self.siz})! + 1, \text{n})$$
$$\wedge\ \mathit{atol}(Z, 0, \$(\text{self.siz})! + 1) = \mathit{app}(N, \mathit{SELF})$$
$$\wedge\ \mathit{intarray}(Z) \wedge 0 \leq \$(\text{self.siz})! + 1 \leq \mathit{a\_size}(Z)$$

and

$$
\begin{array}{ll}
 & lng(pltol(aP(\mathrm{self},\$))) < maxint \wedge INV \\
\Rightarrow & \\
 & \$(\mathrm{self.siz})! + 1 \leq maxint
\end{array}
$$

where *INV* denotes the invariant of interface PLIST in closed form. The proofs of these implication essentially apply definitions and make heavy use of the invariant.

## 4.3.2 Verifying Invariant Properties

In this subsection we illustrate techniques to prove invariant properties. Essentially there are three kinds of invariant properties:

- Absence of side–effects, i.e. that only locations of new objects are affected by the method.
- Environment equivalence for all objects of alien types.
- Class invariants.

The needed proof techniques are illustrated by verifying the properties for methods PLIST::empty and PLIST::append (cf. page 98 and 95 for the program text of these methods). We chose these methods because they create new objects which is the interesting aspect in proving invariant properties, because they depend on the specification of interface ARRAY, and because they are simple enough to focus on the essential aspects.

### Absence of Side–Effects

In most cases, it is fairly simple to prove that a method m does not produce side–effects. The trivial, but most common case is that methods producing no side–effects create objects only by method new and use only methods producing no side–effects. Here, we show the absence of side–effects in a more complex case where the creation of new objects is performed by an imported method, i.e. verification has to rely on the specification of the imported method. As example we prove:

```
meth  append( n: INT ): PLIST
   pre   E<<$
   post  E<<$
```

The proof of this triple illustrates the application of (a) the disjointness property of ARRAY::resize, (b) the equivalence property of ARRAY::set, (c) lemma 3.5.(iv) and (v), and (d) the rearranging technique:

$\{\ lng(pltol(aP(\text{self},\$))) < maxint \wedge INV \wedge E = \$\ \}$
```
iv := self.siz ;
iv := iv.add(1) ;
av := self.arr ;
```
$\{\ 0 \leq \text{iv!} \wedge E = \$\ \}$

$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\downarrow \quad [\![\ \text{rearranging}\ ]\!]$

$\{\ 0 \leq \text{iv!} \wedge alive(X,\$) \wedge E = \$\ \}$

$\overline{\qquad\qquad\qquad\qquad\qquad\qquad}\downarrow \quad [\![\ \text{weakening; elimination of } A\ ]\!]$

$\{\ 0 \leq \text{iv!} \wedge aA(\text{av},\$) = A \wedge alive(X,\$) \wedge E = \$\ \}$

av := `av.resize(iv)`; $[\![$ all specs. of resize, rearranging for no side–effect spec $]\!]$

$\{\ 0 \leq \text{iv!} \wedge aA(\text{av},\$) = a\_resize(A,\text{iv!}) \wedge \neg oreach(X,\text{av},\$) \wedge E \equiv_X \$\ \}$

$\overline{\qquad\qquad\qquad\qquad\qquad\qquad}\uparrow$

$\{\ 0 < \text{iv!} \leq a\_size(aA(\text{av},\$)) \wedge \neg oreach(X,\text{av},\$) \wedge E \equiv_X \$\ \}$
```
av.set(iv,n) ;
```
$\{\ E \equiv_X \$\ \}$

$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\uparrow$

$\{\ \forall X : alive(X,E) \Rightarrow E \equiv_X \$\ \}$

$\Rightarrow$

$\{\ E \ll \$\ \}$

$\Rightarrow \quad [\![\ \text{lemma 3.5.(iv) and (v)}\ ]\!]$

$\{\ \neg isvoid(new(\$,\text{PLIST})) \wedge W = new(\$,\text{PLIST})$
$\quad \wedge\ E \ll \$\langle \text{PLIST}; W.\text{pos} := 0; W.\text{arr} := \text{av}; W.\text{siz} := 0\rangle\ \}$
```
ilv := new PLIST ;
ilv.pos := iv ;
ilv.arr := av ;
ilv.siz := iv ;
result := ilv
```
$\{\ E \ll \$\ \}$

To obtain the side–effect–freeness of PLIST::append the equality $E = \$$ has to be weakened to $E \ll \$$. This can be done be renaming $E$ in the equality to $E'$ (subst–rule), adding $E \ll E'$ to pre– and postcondition (inv–rule), exploiting the transitivity of $\ll$ in the postcondition by a strengthening step, and finally eliminating $E'$.

**Invariance of Alien Objects**

In general, the methods of an interface can manipulate all objects of the environment. As explained in section 4.2.4, we can specify that an interface or some of its methods do not modify objects of alien types. This subsection shows how such an interface property can be derived based on corresponding properties of imported interfaces. As example, we proof that PLIST::empty maintains the $T$–equivalence for alien types:

$$\{\ \mathit{extern}(T, \text{ARRAY\_MOD}) \wedge \mathit{alien}(T, \text{PLIST}) \wedge E \equiv_T \$\ \}\quad \text{PLIST::empty}\ \{\ E \equiv_T \$\ \}$$

The proof contains two typical aspects: 1. It illustrates the application of the lemma 4.9. 2. It demonstrates how specifications of $T$–equivalence for called methods are used. In the example, method PLIST::empty calls method ARRAY::create. To use the specification of ARRAY::create in the proof for PLIST::empty, we need the following implication:

$$\mathit{extern}(T, \text{ARRAY\_MOD}) \wedge \mathit{alien}(T, \text{PLIST}) \ \Rightarrow\ \mathit{alien}(T, \text{ARRAY}) \tag{4.14}$$

The derivation of $\mathit{alien}(T, \text{ARRAY})$ from the premise exploits the fact that ARRAY is private in module LIST_MOD, i.e. property 4.5. As we have not illustrated such derivations so far, we show it in greater detail and prove the following equivalent[13] implication:

$$\mathit{extern}(T, \text{ARRAY\_MOD}) \wedge \mathit{treach}(T, \text{ARRAY}) \ \Rightarrow\ \mathit{treach}(T, \text{PLIST})$$

We consider two cases: 1. $\mathit{treach}(T, \text{OBJECT})$ and 2. $\neg\mathit{treach}(T, \text{OBJECT})$. The assumption of the first case directly implies the conclusion. Thus, we may assume $\neg\mathit{treach}(T, \text{OBJECT})$ in the following proof. To show the above implication, we proof

$$\mathit{extern}(T, \text{ARRAY\_MOD}) \wedge \mathit{treach}_n(T, N, \text{ARRAY}) \ \Rightarrow\ \mathit{treach}(T, \text{PLIST})$$

by induction on N:

**Induction Base:**

$$\begin{array}{ll}
 & \mathit{extern}(T, \text{ARRAY\_MOD}) \wedge \mathit{treach}_n(T, 0, \text{ARRAY}) \\
\Rightarrow & \\
 & \mathit{extern}(T, \text{ARRAY\_MOD}) \wedge T \succeq \text{ARRAY} \\
\Rightarrow & \quad [\![ \text{ all supertypes of ARRAY are known } ]\!] \\
 & \mathit{extern}(T, \text{ARRAY\_MOD}) \wedge (T = \text{OBJECT} \vee T = \text{ARRAY}) \\
\Rightarrow & \quad [\![\ T \neq \text{OBJECT because of } \neg\mathit{treach}(T, \text{OBJECT}) \text{ and} \\
 & \qquad T \neq \text{ARRAY because of } \mathit{extern}(T, \text{ARRAY\_MOD})\ ]\!] \\
 & \mathit{false} \\
\Rightarrow & \\
 & \mathit{treach}(T, \text{PLIST})
\end{array}$$

[13]Recall that $\mathit{alien}(T, ct(T')) \ \Leftrightarrow\ \neg\mathit{treach}(T, ct(T'))$

**Induction Step:**

$$
\begin{array}{ll}
 & \mathit{treach}_n(T, N+1, \mathrm{ARRAY}) \\
\Leftrightarrow & [\![ \text{ definition of } \mathit{treach}_n ]\!] \\
 & \exists A : \mathit{treach}_n(T, N, \mathit{dtyp}(A)) \wedge \mathit{rtyp}(A) \succeq \mathrm{ARRAY} \\
\Rightarrow & \\
 & \exists A : \mathit{treach}_n(T, N, \mathit{dtyp}(A)) \wedge (\mathit{rtyp}(A) = \mathrm{ARRAY} \vee \mathit{rtyp}(A) = \mathrm{OBJECT}) \\
\Rightarrow & [\![ \ \mathit{rtyp}(A) \neq \mathrm{OBJECT} \text{ because otherwise } \mathit{treach}(T, \mathrm{OBJECT}) \ ]\!] \\
 & \exists A : \mathit{treach}_n(T, N, \mathit{dtyp}(A)) \wedge \mathit{rtyp}(A) = \mathrm{ARRAY} \\
\Rightarrow & [\![ \text{ property 4.5 } ]\!] \\
 & \exists A : \mathit{treach}_n(T, N, \mathit{dtyp}(A)) \wedge \mathit{rtyp}(A) = \mathrm{ARRAY} \\
 & \quad \wedge \neg \mathit{extern}(\mathit{dtyp}(\mathrm{A}), \mathrm{LIST\_MOD}) \\
\Rightarrow & [\![ \text{ all concrete interfaces of LIST\_MOD are known } ]\!] \\
 & \exists A : \mathit{treach}_n(T, N, \mathit{dtyp}(A)) \wedge \mathit{rtyp}(A) = \mathrm{ARRAY} \\
 & \quad \wedge (\mathit{dtyp}(\mathrm{A}) = \mathrm{CLIST} \vee \mathit{dtyp}(\mathrm{A}) = \mathrm{PLIST} \vee \mathit{dtyp}(\mathrm{A}) = \mathrm{ARRAY}) \\
\Rightarrow & [\![ \ \nexists A : \mathit{dtyp}(\mathrm{A}) = \mathrm{CLIST} \wedge \mathit{rtyp}(A) = \mathrm{ARRAY} \ ]\!] \\
 & \mathit{treach}_n(T, N, \mathrm{ARRAY}) \vee \mathit{treach}_n(T, N, \mathrm{PLIST}) \\
\Rightarrow & [\![ \text{ induction hypothesis } ]\!] \\
 & \mathit{treach}(T, \mathrm{PLIST})
\end{array}
$$

Beside the proven property 4.14, the proof of the $T$–equivalence for PLIST::empty uses lemma 4.9:

$\{ \ \mathit{extern}(T, \mathrm{ARRAY\_MOD}) \wedge \mathit{alien}(T, \mathrm{PLIST}) \wedge E \equiv_T \$ \ \}$

$\Rightarrow \ [\![ \text{ property 4.14 } ]\!]$

$\{ \ 0 \leq 0 \wedge \mathit{alien}(T, \mathrm{PLIST}) \wedge \mathit{alien}(T, \mathrm{ARRAY}) \wedge E \equiv_T \$ \ \}$

`av := ARRAY :: create(0);` $[\![$ cf. subsection 4.2.4.2 $]\!]$

$\{ \ \mathit{alien}(T, \mathrm{PLIST}) \wedge E \equiv_T \$ \ \}$

$\Rightarrow \ [\![ \text{ lemma 4.9 } ]\!]$

$\{ \ \exists W : W = \mathit{new}(\$, \mathrm{PLIST})$

$\quad \wedge \ E \equiv_T \$\langle \mathrm{PLIST}; W.\mathrm{pos} := \mathrm{iv}; W.\mathrm{arr} := \mathrm{av}; W.\mathrm{siz} := \mathrm{iv} \rangle \ \}$

```
ilv := new PLIST ;
ilv.pos := 0 ;
ilv.arr := av ;
ilv.siz := 0 ;
result := ilv
```

$\{ \ E \equiv_T \$ \ \}$

## Class Invariants

The last kind of invariance properties are class invariants. In several aspects, the verification of class invariants differs from proving the absence of side–effects or the equivalence for alien objects. Roughly speaking, a method m has the latter properties if all methods called within the body of m have the properties. For the verification of class invariants this is not sufficient, because class invariants state as well properties of those objects that are created and modified in the method under consideration.

Class invariants are universally quantified over all objects. The canonical proof technique is to use a case distinction over the quantification range as will be illustrated in the following. As example, we prove that PLIST::empty maintains the conjunction $inv(X,E)$ of the class invariant predicates for CLIST and PLIST[14]:

$$\begin{array}{lll} inv(X,E) \Leftrightarrow_{def} & & isvoid(X) \vee \neg alive(X,E) \\ \vee \; ( & (typ(X) = \text{CLIST} \;\Rightarrow\; wfC(X,E) \wedge lng(aC(X,E)) \leq maxint) & \\ & \wedge \; (typ(X) = \text{PLIST} \;\Rightarrow\; wfP(X,E) \wedge lng(pltol(aP(X,E))) \leq maxint) \; ) & \end{array}$$

We have to prove:

$\{\ \forall X : inv(X,\$)\ \}$
```
meth PLIST::empty(): PLIST
```
$\{\ \forall X : inv(X,\$)\ \}$

For method PLIST::empty it is suffices to show that the invariant is maintained for all X individually, i.e. we show $\{\ inv(X,\$)\ \}$ empty $\{\ inv(X,\$)\ \}$ from which the above triple can be derived as demonstrated together with the all–rule (cf. section 3.3). To prove that the invariant is maintained we distinguish four cases:

A: $\{\ typ(X) \not\preceq \text{LIST}\ \}$ meth PLIST::empty $\{\ inv(X,\$)\ \}$

B: $\{\ inv(X,\$) \wedge alive(X,\$)\ \}$ meth PLIST::empty $\{\ inv(X,\$)\ \}$

C: $\{\ \neg alive(X,\$) \wedge X \neq new(\$,\text{PLIST}) \wedge typ(X) \preceq \text{LIST}\ \}$
meth PLIST::empty $\{\ inv(X,\$)\ \}$

D: $\{\ X = new(\$,\text{PLIST})\ \}$ meth PLIST::empty $\{\ inv(X,\$)\ \}$

By adding $inv(X,\$)$ as a conjunct to the preconditions and applying the disjunction rule we can derive the desired proof obligations. The four cases are treated in turn.

**Case A:** Based on the triple $\{$ TRUE $\}$ empty $\{$ TRUE $\}$ which is simple to show, the proof obligation for case A can be derived as follows:

$\{\ \text{TRUE} \wedge typ(X) \not\preceq \text{LIST}\ \}$
```
meth PLIST::empty
```
$\{\ \text{TRUE} \wedge typ(X) \not\preceq \text{LIST}\ \}$
$\Rightarrow$
$\{\ inv(X,\$)\ \}$

[14]To keep things simple, we do not consider sharing properties in the invariant and assume that type LIST is closed.

**Case B:** The essential property to prove this case is the $X$–equivalence of the invariant (cf. property 4.11):

$$
\begin{array}{l}
\quad \{\ inv(X,\$) \wedge alive(X,\$)\ \} \\
\quad \Rightarrow \\
\quad \{\ \exists E : inv(X,\$) \wedge alive(X,\$) \wedge E = \$\ \} \\
\quad \rule{8cm}{0.4pt} \downarrow \\
\quad \{\ inv(X,E) \wedge alive(X,E) \wedge E = \$\ \} \\
\texttt{meth PLIST::empty} \\
\quad \{\ inv(X,E) \wedge alive(X,E) \wedge E \ll \$\ \} \\
\quad \Rightarrow\ [\![\ \text{lemma 3.5.(iii)}\ ]\!] \\
\quad \{\ inv(X,E) \wedge E \equiv_X \$\ \} \\
\quad \Rightarrow\ [\![\ \text{property 4.11}\ ]\!] \\
\quad \{\ inv(X,\$)\ \} \\
\quad \rule{8cm}{0.4pt} \uparrow \\
\quad \{\ inv(X,\$)\ \}
\end{array}
$$

**Case C:** The derivation of case C is based on the property

$$\{\ \mathbf{R} \wedge \neg alive(X,\$)\ \}\ \texttt{ARRAY} :: \texttt{create}\ \{\ alive(X,\$)\ \Rightarrow\ typ(X) = \text{ARRAY}\ \}$$

proved in subsection 4.2.3.1. From that property we can derive

$$\{\ \mathbf{R} \wedge \neg alive(X,\$) \wedge typ(X) \preceq \text{LIST}\ \}\ \texttt{ARRAY} :: \texttt{create}\ \{\ \neg alive(X,\$)\ \}$$

which is the central triple in the following proof outline:

$$
\begin{array}{l}
\quad \{\ \neg alive(X,\$) \wedge typ(X) \preceq \text{LIST} \wedge X \neq new(\$,\text{PLIST})\ \} \\
\texttt{av} := \texttt{ARRAY} :: \texttt{create}(0);\quad [\![\ \text{see above and local creation property (cf. p. 111)}\ ]\!] \\
\quad \{\ \neg alive(X,\$) \wedge X \neq new(\$,\text{PLIST})\ \} \\
\quad \Rightarrow \\
\quad \{\ \neg alive(X, \$\langle \text{PLIST}; W.\text{pos} := 0; W.\text{arr} := \text{av}; W.\text{siz} := 0\rangle) \\
\quad\quad \wedge\ W = new(\$,\text{PLIST})\ \} \\
\texttt{ilv := new PLIST ;} \\
\texttt{ilv.pos := 0 ;} \\
\texttt{ilv.arr := av;} \\
\texttt{ilv.siz := 0 ;} \\
\texttt{result := ilv} \\
\quad \{\ \neg alive(X,\$)\ \} \\
\quad \Rightarrow \\
\quad \{\ inv(X,\$)\ \}
\end{array}
$$

**Case D:** In this case, the invariant has to be established for the newly created object. Thus, it is similar to the verification of functional properties as demonstrated in section 4.3.1. The proof illustrates in particular the need of annotations specifying which objects are created by a method (cf. subsection 4.2.4.2), demonstrates the use of liveness properties of variables and shows where $X$–equivalence of abstraction functions is required.

$\{\ X = new(\$, \mathrm{PLIST})\ \}$

$\Rightarrow$

$\{\ 0 \leq 0 \wedge \mathrm{PLIST} \neq \mathrm{ARRAY} \wedge X = new(\$, \mathrm{PLIST})\ \}$

`av` $:=$ `ARRAY` $::$ `create(0);` $[\![$ local creation property $]\!]$

$\{\ a\_size(aA(\mathrm{av}, \$) = 0 \wedge X = new(\$, \mathrm{PLIST})\ \}$

$\Rightarrow\ [\![$ all variables are alive; cf. lemma 3.9 $]\!]$

$\{\ alive(\mathrm{av}, \$) \wedge 0 \leq a\_size(aA(av, \$)) \wedge X = new(\$, \mathrm{PLIST})\ \}$

———————————————————— $\downarrow$ $[\![$ ex–rule, strengthening $]\!]$

$\{\ alive(\mathrm{av}, \$) \wedge 0 \leq a\_size(aA(\mathrm{av}, \$)) \wedge X = new(\$, \mathrm{PLIST})$
$\wedge\ E = \$\langle \mathrm{PLIST}; X.\mathrm{pos} := 0; X.\mathrm{arr} := \mathrm{av}; X.\mathrm{siz} := 0 \rangle)\ \}$

$\Rightarrow\ [\![$ env11, $X$–equivalence of $aA$, definitions $]\!]$

$\{\ typ(X) = ct(\mathrm{PLIST}) \wedge \neg isvoid(X) \wedge X = new(\$, \mathrm{PLIST})$
$\wedge\ 0 \leq pos(X, E) \leq siz(X, E) \leq a\_size(aA(arr(X, E), E))$
$\wedge\ intarray(arr(X, E)) \wedge lng(pltol(aP(X, E))) \leq maxint$
$\wedge\ E = \$\langle \mathrm{PLIST}; X.\mathrm{pos} := 0; X.\mathrm{arr} := \mathrm{av}; X.\mathrm{siz} := 0 \rangle)\ \}$

$\Rightarrow\ [\![$ definition of $inv$(X,E) and $wfP$ $]\!]$

$\{\ \neg isvoid(new(\$, \mathrm{PLIST})) \wedge inv(X, E) \wedge X = new(\$, \mathrm{PLIST})$
$\wedge\ E = \$\langle \mathrm{PLIST}; X.\mathrm{pos} := 0; X.\mathrm{arr} := \mathrm{av}; X.\mathrm{siz} := 0 \rangle)\ \}$

```
ilv := new PLIST ;
ilv.pos := 0 ;
ilv.arr := av ;
ilv.siz := 0 ;
result := ilv
```

$\{\ inv(X, \$)\ \}$

———————————————————— $\uparrow$

$\{\ inv(X, \$)\ \}$

### 4.3.3 Exploiting Sharing Properties

Subsection 4.2.4.3 discussed the specification of sharing behaviour. As example it provided an interface specification for the abstract interface LIST. The sharing behaviour of interface LIST is implementation–independent. In particular, it applies to class PLIST and class CLIST. This section demonstrates that sharing specifications can be used for verification in the same way as other interface specification parts. As a small example we prove that the following program fragment (cf. chapter 2)

```
tmp :=  l.first() ;
l.updfst( l.rest().first() ) ;
l.rest(). updfst( tmp )
```

swaps the first two elements of list l, if l has at least two elements. The proof outline for the swapping property is based on an AICKL*–version of the above fragment (the invariant of interface LIST is kept implicit):

$\{\ aL(l,\$) = app(M, app(N, RL))\ \}$

`tmp := l.first() ;`

$\{\ aL(\mathrm{l},\$) = app(M, app(N, RL)) \wedge \mathrm{tmp!} = M\ \}$

`lrest := l.rest() ;`

$\{\ dirpart(\mathrm{lrest},\mathrm{l},\$) \wedge aL(\mathrm{lrest},\$) = app(N, RL)$
$\wedge\ aL(\mathrm{l},\$) = app(M, app(N, RL)) \wedge \mathrm{tmp!} = M\ \}$

`lrfst := lrest.first() ;`

$\{\ dirpart(\mathrm{lrest},\mathrm{l},\$) \wedge aL(\mathrm{lrest},\$) = app(N, RL)$
$\wedge\ aL(\mathrm{l},\$) = app(M, app(N, RL)) \wedge \mathrm{lrfst!} = N \wedge \mathrm{tmp!} = M\ \}$

$\Rightarrow\ [\![\ \text{property 4.13}\ ]\!]$

$\{\ dirpart(\mathrm{lrest},\mathrm{l},\$)$
$\wedge\ \neg part(\mathrm{l},\mathrm{lrest},\$) \wedge aL(\mathrm{lrest},\$) = app(N, RL)$
$\wedge\ aL(\mathrm{l},\$) = app(M, app(N, RL)) \wedge \mathrm{lrfst!} = N \wedge \mathrm{tmp!} = M\ \}$

`l.updfst( lrfst ) ;`

$\{\ dirpart(\mathrm{lrest},\mathrm{l},\$) \wedge aL(\mathrm{lrest},\$) = app(N, RL)$
$\wedge\ aL(\mathrm{l},\$) = app(N, app(N, RL)) \wedge \mathrm{tmp!} = M\ \}$

$\Rightarrow$

$\{\ part(\mathrm{lrest},\mathrm{l},\$) \wedge aL(\mathrm{lrest},\$) = app(N, RL)$
$\wedge\ aL(\mathrm{l},\$) = conc(app(N, empt), app(N, RL)) \wedge \mathrm{tmp!} = M\ \}$

`lrest . updfst( tmp ) ;`

$\{\ aL(\mathrm{l},\$) = conc(app(N, empt), app(M, rst(app(N, RL)))) \wedge \mathrm{tmp!} = M\ \}$

$\Rightarrow$

$\{\ aL(\mathrm{l},\$) = app(N, app(M, RL))\ \}$

As methods first, rest, and updfst maintain the *part*–relation on lists, the whole fragment maintains this property.

# Chapter 5

# Conclusions

*"Born in the ice–blue waters of the festooned Norwegian coast; amplified along the much grayer range of the Californian Pacific; viewed by some as a typhoon, by some as a tsunami, and by some as a storm in a teacup — a tidal wave is reaching the shores of the computing world."* Bertrand Meyer in [Mey88]

By its advocators, object–oriented programming is considered to be a "proper" progress compared to classical procedural programming. It is said and written that object–oriented programs provide more structure, correspond closer to the world they model, have better designed interfaces, are easier to extend and adapt, are more appropriate for reuse, and are safer because of encapsulation.

Even if programming is embedded in an informal context, it is essentially a formal activity. Thus, it could be expected that the informally stated advantages of object–orientation correspond to advantages for the formal specification and verification of programs. This expectation was one of the major *motivations* for the development of the integrated specification and verification framework presented in this thesis.

In these conclusions, we summarize what has been achieved, discuss in which respect our expectations have become true, and sketch lines of future research.

**Summary** The thesis integrates interface specification and verification of object–oriented programs within a formal framework and develops techniques to specify data representation properties in an implementation–independent way. The integration method and the developed techniques are applied to an object–oriented kernel language that supports recursive classes, recursive methods, separation of interfaces and implementations, subtyping, encapsulation, and a primitive module concept.

The integration of interface specification and program verification is based on the axiomatic semantics of the underlying programming language. The axiomatic semantics is given in form of Hoare triples. Hoare triples are used to give a precise meaning to interface specifications (including invariants) and they provide the basis for the programming logic.

Syntactically, an interface specification of a type and its methods consists of (a) an invariant for the type and (b) a requires clause and a set of pre–postcondition pairs for each of its methods. An important contribution of the integrated framework is that it provides for the first time a formal and appropriate meaning of invariants. An invariant has to remain invariant under the execution of all publicly accessible methods. As we demonstrated in subsection 4.2.3.2, it is in general not sufficient for program verification to require that an invariant of a type T has to remain invariant under the execution of T's methods.

The presented approach to interface specification supports not only the specification of functional method behaviour. It enables as well to specify data representation properties and properties of the object environment. These additional aspects are a prerequisite to make the framework applicable to a wide class of languages and programs. Most object–oriented programs use sharing of data representation and destructive updates to achieve efficiency. We developed techniques to specify these aspects in an implementation–independent way. Implementation–independency is desirable to enable modifications of implementations without having to change the interface specifications and the proofs using them. Implementation–independency is needed to specify sharing and invariance properties of types that have several subtypes with different implementations.

The developed programming logic differs in three aspects from a partial correctness Hoare logic for programs with recursive procedures. 1. To capture linked object structures it uses a global variable. The global variable holds values of an abstract data type modeling object environments. Operations to access and modify this variable within programs are specified by program axioms. 2. The programming logic supports subtyping and dynamic method selection[1]. The basic idea underlying this extension is to relate the properties of subtype methods to the properties of supertype methods by a proof rule. 3. The programming logic was designed in such a way that the absence of runtime errors (illegal dereferencing, arithmetic overflow) can be proved. As these errors are a frequent source of program misbehaviour, we decided to incorporate this feature although it makes the application of the logic a bit more complex.

**Object–Oriented Programming and Formal Methods** This thesis was partly motivated by the expectation that object–oriented models and programming constructs could simplify and improve the formal specification and verification of programs. What are the conclusions with respect to this expectation? The positive part of the answer is that object–oriented concepts may be helpful, are useful or even necessary for making program specification and verification practical. We briefly sketch four aspects:

- Subclassing may be helpful to construct a new class from a given one in such a way that a semantic subtype relation between the two classes can be established. The proof of the semantic subtype relation need not be automatic but a careful design of the subclass concept can simplify such proofs. In most of the object–oriented languages that we have studied, it seems more difficult for us to prove this semantic subtype relation for a derived subclass than for a newly implemented class (i.e. by using only subtyping and not subclassing; cf. subsection 4.1.1.3). But in future languages this may be different.

[1]Dynamic method selection is often called dispatching.

- The structuring into types with well–defined interfaces is useful, in particular as a methodological mechanism for designing program invariants.
- Encapsulation is needed to guarantee or at least to make it easy to prove that valid properties of a type or module cannot be invalidated by putting the type or module into a larger program context. In particular, integrity constraints on data representations can only be maintained if access to these representations is restricted.
- Subtyping is very useful as a mechanism to structure programs and proofs (cf. the verification of the sorting method in section 4.3.1 that captures two list implementations in one proof). But it can be dangerous as it may cause implicit modifications to related types (cf. subsection 4.2.3.2).

The negative part of the answer to the question above is that the methods and techniques presented in this thesis only provide necessary foundations on the way towards a theory for logic–based object–oriented programming. Many important problems remain to be solved. Hard engineering work to improve existing languages and to construct powerful tools remains to be done. Future research directions are sketched in the following paragraph.

**Future Research** This paragraph outlines two research directions that we consider a direct consequence of the experiences made while writing this thesis. Just as the vision underlying this thesis is not new, these research directions are in general well–known. This thesis may contribute to reformulate or focus them.

**Logic–Based Programming Environments** Object–orientation may help to structure proofs for larger programs. However, verifying parts or modules of object–oriented programs is almost as strenuous as proving procedural programs with pointer structures. Without powerful machine support, practical applications of formal methods at the program level will remain unrealistic. Just as efficient compilers played the essential role for using high–level programming languages, the development of logic–based programming environments, i.e. of software tools supporting the specification, verification, and composition of program components, is crucial for establishing logic–based program construction. As logic–based programming environments are very complex language–specific software systems, it is important to develop generation techniques for the implementation of their language–specific parts in order to keep pace with the development of programming languages.

**Interface Design and Specification** Programming rarely means to write entire programs from scratch. To a large extent, it is the activity of interfacing and controlling existing modules or systems by instantiating parameters, writing program fragments to link module and system interfaces, and adapting library packages. In the future, most software products will enable to plug in extensions, i.e. further functionality or customization packages. E.g. a document processing system may be costumized to a variety of backends or may support plug–ins for different picture and graphical formats to be placed into the document. Beside type checking there is nowadays no mechanically supported, generally accepted technique to formulate or establish properties of composed programs or of interfaces for plug–in modules. To improve this situation we need better answers to the following questions:

- Module extension: How can extensions to modules (and types) be supported such that the extended modules maintain the semantical properties of the base module? Or, to put it more specific for inheritance: How does subclassing has to be designed in order to make it simple to prove that a subclass is a semantical subtype of the superclass?

- Parameterization: A plug–in module is a complex program parameter that has dynamic behaviour of its own. How can we specify the interface requirements of such parameters? Or, to put it more specific for object–oriented programming languages: How do we specify interface requirements for parameters that are classes?

- Interface Specification Design: An interface specification should describe all relevant properties of a module interface such that the correctness of user programs can be established without knowing the implementation of the module. How do we design interface specifications to reach this goal? This question has a methodological aspect — what are the steps towards a good interface specification? — and a completeness aspect: If some interface properties are forgotten in the specification, it may be possible that other properties cannot be exploited for the verification of user programs (e.g. a sharing specification can only be used if some method establishes a sharing property; cf. subsection 4.2.4.3).

All these aspects become even more important in connection with the application of software in distributed environments.

"The vision underlying this work is that efficient programs can be constructed by interfacing well–defined software components in such a way that the correctness and properties of the constructed programs can be derived from the specifications of the components" (cf. introduction). This vision might sound as unrealistic to software engineers and programmers today as spacelabs might have sounded to people of the last century who could not even imagine an aeroplane. This work aimed at making the vision a bit more realistic.

# Appendix A

# Models, Summaries, Proofs, Examples

## A.1 Model for the Environment Axioms

The axioms env1—env13 specifying the environment operations are fairly complex. In order to show their consistency, we provide an algebra for these operations and prove that this algebra is a model. We assume an algebra $\mathcal{O}$ with carrier sets for sorts *Integer*, *Int*, *Boolean*, *ObjId*, *AttId*, *TypId*, *Type*, *Object*, *Location* and operations on these sorts according the definitions in section 3.1.2. The carrier sets and operations in the algebra are written in roman font, i.e. we have carrier sets Integer, etc. and operations typ, isvoid, etc.. Recall from section 3.1.2 that ObjId is an infinite set. To keep things simple, we use natural numbers as object identifiers, in particular we assume a successor function and a comparison predicate $<$. The algebra $\mathcal{O}$ is extended by the additional carrier set ObjEnv. ObjEnv is a subset of $(\mathrm{TypId} \to \mathrm{ObjId}) \times (\mathrm{Location} \to \mathrm{Object})$ where the first component records for each declared type the object identifiers for the object to be created next and the second component maps locations to objects. Giving a function $\alpha$ from TypId to ObjId, we define the set of living objects $\mathrm{Liv}(\alpha)$ as follows:

$$\mathrm{Liv}(\alpha) =_{def} \{X \in \mathrm{Object} \mid X \text{ is static or } X = \mathrm{object}(T, O) \text{ where } O < \alpha(T)\ \}$$

Based on this abbreviation, we formulate *two conditions* that have to be satisfied by the location mapping:

$(\alpha, \eta) \in \mathrm{ObjEnv} \ \Leftrightarrow\ _{def}$
$\eta(K)$ is in $\mathrm{Liv}(\alpha)$ for all locations $K$ (condition 1)
if $\mathrm{obj}(K)$ is not in $\mathrm{Liv}(\alpha)$, then $\eta(K) = \mathrm{init}(\mathrm{ltyp}(K))$ (condition 2)

The set ObjEnv is non–empty, e.g. let $\alpha$ map all type identifiers to zero and let $\eta$ be such that $\eta(L)$ equals the initial object of the location type of $L$ for all locations $L$.

The operations on ObjEnv are defined as follows:

$(\alpha,\eta)\texttt{<}L\texttt{:=}X\texttt{>} =_{def}$
$\quad(\ \alpha,\ \lambda K\,.\,\text{if } X\in\text{Liv}(\alpha) \text{ and } \text{obj}(L)\in\text{Liv}(\alpha) \text{ and } K=L \text{ then } X \text{ else } \eta(K) \text{ fi }\ )$
$(\alpha,\eta)\texttt{<}T\texttt{>} =_{def}$
$\quad(\ \lambda S\,.\,\text{if } S=T \text{ then succ}(\alpha(T)) \text{ else } \alpha(S) \text{ fi},\ \eta\ )$
$(\alpha,\eta)\texttt{(}L\texttt{)} =_{def} \eta(L)$
$\text{alive}(X,(\alpha,\eta)) =_{def} (X\in\text{Liv}(\alpha))$
$\text{new}((\alpha,\eta),T) =_{def} \text{object}(\text{T},\alpha(T))$

For well–definedness, we have to show that environment updates and object creations yield elements in ObjEnv. It is easy to see that the location mapping satisfies condition 1 and 2 after an object creation. Thus, it remains to show that an updated location mapping satisfies condition 1 and 2. We only have to check it for the updated location $L$, the other case is trivial. The condition of the if–then–else yields $X\in\text{Liv}(\alpha)$ and $\text{obj}(L)\in\text{Liv}(\alpha)$. The first conjunct implies condition 1 for $L$ and the second implies condition 2 for $L$.

To show that the algebra defined above is a model for the environment axioms, let $(\alpha,\eta),(\alpha_1,\eta_1),(\alpha_2,\eta_2)\in\text{ObjEnv}$, $X,X1\in\text{Object}$, $L,L1,L2\in\text{Location}$ and $T\in\text{TypId}$. We consider each axiom in turn:

1. We may assume $L1\neq L2$:

$$\begin{aligned}
& (\alpha,\eta)\texttt{<}L1\texttt{:=}X\texttt{>(}L2\texttt{)}\\
= \quad & [\![\text{ applying definitions }]\!]\\
& (\lambda K.\text{if } X\in\text{Liv}(\alpha) \text{ and } \text{obj}(L1)\in\text{Liv}(\alpha) \text{ and } K=L1 \text{ then } X\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \eta(K) \text{ fi})(L2)\\
= \quad & [\![\ L1\neq L2\ ]\!]\\
& \eta(L2)\\
= \quad & \\
& (\alpha,\eta)\texttt{(}L2\texttt{)}
\end{aligned}$$

2. We may assume $\text{alive}(\text{obj}(L),(\alpha,\eta))$ and $\text{alive}(X,(\alpha,\eta))$, i.e. $\text{obj}(L)\in\text{Liv}(\alpha)$ and $X\in\text{Liv}(\alpha)$:

$$\begin{aligned}
& (\alpha,\eta)\texttt{<}L\texttt{:=}X\texttt{>(}L\texttt{)}\\
= \quad & [\![\text{ applying definitions }]\!]\\
& (\lambda K.\text{if } X\in\text{Liv}(\alpha) \text{ and } \text{obj}(L)\in\text{Liv}(\alpha) \text{ and } K=L \text{ then } X\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \eta(K) \text{ fi})(L)\\
= \quad & [\![\ \text{obj}(L)\in\text{Liv}(\alpha) \text{ and } X\in\text{Liv}(\alpha)\ ]\!]\\
& X
\end{aligned}$$

3. We may assume $X \notin \mathrm{Liv}(\alpha)$:

$$
\begin{aligned}
& (\alpha, \eta)\texttt{<}L\texttt{:=}X\texttt{>} \\
= \quad & [\![ \text{ applying definitions } ]\!] \\
& (\ \alpha,\ \lambda K.\text{if } X \in \mathrm{Liv}(\alpha) \text{ and } \mathrm{obj}(L) \in \mathrm{Liv}(\alpha) \text{ and } K = L \text{ then } X \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \eta(K) \text{ fi } ) \\
= \quad & [\![\ X \notin \mathrm{Liv}(\alpha)\ ]\!] \\
& (\ \alpha,\ \lambda K.\eta(K)\ ) \\
= \quad & \\
& (\alpha, \eta)
\end{aligned}
$$

4. We may assume $\mathrm{obj}(L) \notin \mathrm{Liv}(\alpha)$, i.e. condition 2 of the ObjEnv–definition yields $\eta(L) = \mathrm{init}(\mathrm{ltyp}(L))$:

$$
\begin{aligned}
& (\alpha, \eta)\texttt{(}L\texttt{)} \\
= \quad & [\![ \text{ applying definitions } ]\!] \\
& \eta(L) \\
= \quad & [\![\ \eta(L) = \mathrm{init}(\mathrm{ltyp}(L))\ ]\!] \\
& \mathrm{init}(\mathrm{ltyp}(L))
\end{aligned}
$$

5.
$$
\begin{aligned}
& (\alpha, \eta)\texttt{<}T\texttt{>(}L\texttt{)} \\
= \quad & [\![ \text{ applying definitions } ]\!] \\
& (\ \lambda S\,.\text{ if } S = T \text{ then } \mathrm{succ}(\alpha(T)) \text{ else } \alpha(S) \text{ fi },\ \eta\ )\texttt{(}L\texttt{)} \\
= \quad & \\
& \eta(L) \\
= \quad & \\
& (\alpha, \eta)\texttt{(}L\texttt{)}
\end{aligned}
$$

6. Direct from the definitions.

7.
$$
\begin{aligned}
& \mathrm{Liv}(\lambda S.\text{if } S = T \text{ then } \mathrm{succ}(\alpha(T)) \text{ else } \alpha(S) \text{ fi}) \\
= \quad & \\
& \{X \in \text{Object} \mid X \text{ is static or } X = \mathrm{object}(S, O) \\
& \qquad\qquad \text{where } O < \alpha(S) \text{ or } (\ S = T \text{ and } O = \alpha(S)\ )\ \} \\
= \quad & \\
& \mathrm{Liv}(\alpha)\ \cup \{\ \mathrm{object}(T, \alpha(T))\ \}
\end{aligned}
$$

8. Follows from condition 1 in the ObjEnv definition.

9. $\mathrm{Liv}(\alpha)$ contains all static objects per definitionem.

10. $\mathrm{object}(T, \alpha(T)) \ \notin\ \mathrm{Liv}(\alpha)$.

11. $\mathrm{typ}(\mathrm{object}(T, \alpha(T))) = \mathrm{ct}(T)$.

12.
$$\begin{aligned}
& \mathrm{object}(T, \alpha_1(T)) = \mathrm{object}(T, \alpha_2(T)) \\
\Leftrightarrow \quad & \llbracket \text{ Object is term generated } \rrbracket \\
& \alpha_1(T) = \alpha_2(T) \\
\Leftrightarrow \quad & \\
& \quad \mathrm{Liv}(\alpha_1) \cap \{X \in \mathrm{Object} \mid \mathrm{typ}(X) = \mathrm{ct}(T)\} \\
& = \mathrm{Liv}(\alpha_2) \cap \{X \in \mathrm{Object} \mid \mathrm{typ}(X) = \mathrm{ct}(T)\}
\end{aligned}$$

13. Simple calculation.

## A.2 Summary of Programming Logic

This appendix summarizes the programming logic for AICKL and AICKL*. Axioms and rules are presented according to the restricted syntax explained in section 3.1.3. In particular, they make no use of the prestate–operator:

equ–axiom:

$$\vdash \{ \text{ self} = S \wedge \text{p} = P \wedge \$ = E \ \} \\ \text{meth T::equ(p)} \ \ \{ \text{ result} = \mathit{bool}(S = P) \wedge \$ = E \ \}$$

idt–axiom:

$$\vdash \{ \text{ self} = S \wedge \$ = E \ \} \ \ \text{meth T::idt()} \ \ \{ \text{ result} = S \wedge \$ = E \ \}$$

new–axiom:

$$\vdash \{ \ \$ = E \ \} \ \ \text{meth T::new()} \ \ \{ \text{ result} = \mathit{new}(E, \mathrm{T}) \wedge \$ = E\langle \mathrm{T} \rangle \ \}$$

add–axiom:

$$\vdash \{ \text{ self} = S \wedge \text{p} = P \wedge \$ = E \wedge \mathit{minint} \leq \mathit{aI}(S) + \mathit{aI}(P) \leq \mathit{maxint} \ \} \\ \text{meth INT::add(p)} \ \ \{ \text{ result} = \mathit{int}(\mathit{aI}(S) + \mathit{aI}(P)) \wedge \$ = E \ \}$$

less–axiom:

$$\vdash \{ \text{ self} = S \wedge \text{p} = P \wedge \$ = E \ \} \\ \text{meth INT::less(p)} \ \ \{ \text{ result} = \mathit{bool}(\mathit{aI}(S) < \mathit{aI}(P)) \wedge \$ = E \ \}$$

not–axiom:

$$\vdash \{ \text{ self} = S \wedge \$ = E \ \} \\ \text{meth BOOL::not()} \ \ \{ \text{ result} = \mathit{bool}(\neg \mathit{aB}(S)) \wedge \$ = E \ \}$$

read–att–axiom:

$$\vdash \{ \text{ self} = S \wedge \$ = E \wedge \neg \mathit{isvoid}(S) \wedge \mathit{typ}(S) = \mathrm{T} \ \} \\ \text{meth T::read\_att()} \ \ \{ \text{ result} = E(S.\text{att}) \wedge \$ = E \ \}$$

write–att–axiom:

$$\vdash \{\ \mathrm{self} = S \wedge \mathrm{p} = P \wedge \$ = E \wedge \neg \mathit{isvoid}(S) \wedge \mathit{typ}(S) = \mathrm{T}\ \}$$
$$\mathrm{meth\ T::write\_att(\ p)}\quad \{\ \$ = E\langle S.\mathrm{att} := P\rangle\ \}$$

cast–axiom:

$$\vdash \{\ \mathrm{self} = S \wedge \$ = E \wedge \mathit{typ}(\mathrm{self}) \preceq \mathrm{T}\ \}$$
$$\mathrm{meth\ OBJECT::T()}\quad \{\ \mathrm{result} = S \wedge \$ = E\ \}$$

while-rule:

$$\frac{\mathcal{A} \vdash \{\ aB(\mathrm{EXP}) \wedge \mathbf{P}\ \}\ \mathrm{STAT}\ \{\ \mathbf{P}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{while\ EXP\ do\ STAT\ end}\ \{\ \neg aB(\mathrm{EXP}) \wedge \mathbf{P}\ \}}$$

if–rule:

$$\frac{\begin{array}{l}\mathcal{A} \vdash \{\ aB(\mathrm{EXP}) \wedge \mathbf{P}\ \}\ \mathrm{STAT1}\ \{\ \mathbf{Q}\ \}\\ \mathcal{A} \vdash \{\ \neg aB(\mathrm{EXP}) \wedge \mathbf{P}\ \}\ \mathrm{STAT2}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{if\ EXP\ then\ STAT1\ else\ STAT2\ end}\ \{\ \mathbf{Q}\ \}}$$

seq-rule:

$$\frac{\begin{array}{l}\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{STAT1}\ \{\ \mathbf{Q}\ \}\\ \mathcal{A} \vdash \{\ \mathbf{Q}\ \}\ \mathrm{STAT2}\ \{\ \mathbf{R}\ \}\end{array}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \mathrm{STAT1}\ ;\ \mathrm{STAT2}\ \ \{\ \mathbf{R}\ \}}$$

call–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \mathrm{meth\ T::m}(\mathrm{p}_1,\ldots,\mathrm{p}_z)\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[\mathrm{E}_0/\mathrm{self},\mathrm{E}_1/\mathrm{p}_1,\ldots,\mathrm{E}_z/\mathrm{p}_z]\ \}\ \mathrm{v} := \mathrm{E}_0\ .\ \mathrm{m}(\mathrm{E}_1,\ldots,\mathrm{E}_z)\ \{\ \mathbf{Q}[\mathrm{v}/\mathrm{result}]\ \}}$$

var–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{v} := \mathrm{E}_0\ .\ \mathrm{m}(\mathrm{E}_1,\ldots,\mathrm{E}_z)\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[\mathrm{w}/Z]\ \}\ \mathrm{v} := \mathrm{E}_0\ .\ \mathrm{m}(\mathrm{E}_1,\ldots,\mathrm{E}_z)\ \{\ \mathbf{Q}[\mathrm{w}/Z]\ \}}$$

rec–rule:

$$\frac{\begin{array}{l}\{\ \mathbf{P}\ \}\ \ \mathrm{meth\ T::m}(\mathrm{p}_1,\ldots,\mathrm{p}_z)\ \{\ \mathbf{Q}\ \},\mathcal{A}\\ \quad \vdash \{\ \mathbf{P} \wedge \bigwedge_i \mathrm{v}_i = \mathit{init}(\mathrm{TV}_i)\ \}\ \mathrm{BODY(T::m)}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \mathrm{meth\ T::m}(\mathrm{p}_1,\ldots,\mathrm{p}_z)\ \{\ \mathbf{Q}\ \}}$$

rec*–rule:

$$\frac{\begin{array}{l}\{\ \mathbf{P}\ \}\ \ \text{meth T::m}(\text{p}_1,\ldots,\text{p}_z)\ \{\ \mathbf{Q}\ \}\ ,\ \mathcal{A}\\ \quad\vdash\ \{\ \mathbf{P}\wedge\ \bigwedge_i \mathit{typ}(\text{v}_i)\preceq\text{TV}_i\wedge\ \bigwedge_i \mathit{static}(\text{v}_i)\ \}\ \text{BODY(T::m)}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A}\vdash\ \{\ \mathbf{P}\ \}\ \ \text{meth T::m}(\text{p}_1,\ldots,\text{p}_z)\ \{\ \mathbf{Q}\ \}}$$

subtype–rule:

$$\frac{\text{T}'\preceq\text{T}\ ,\ \mathcal{A}\vdash\ \{\ \mathit{typ}(\text{self})\preceq\text{T}'\wedge\mathbf{P}\ \}\ \text{meth T'::m}\ \{\ \mathbf{Q}\ \}}{\begin{array}{l}\{\ \mathit{typ}(\text{self})\preceq\text{T}\wedge\mathit{typ}(\text{self})\not\preceq\text{T}'\wedge\mathbf{P}\ \}\ \text{meth T::m}\ \{\ \mathbf{Q}\ \}\ ,\ \mathcal{A}\\ \ \vdash\ \{\ \mathit{typ}(\text{self})\preceq\text{T}\wedge\mathbf{P}\ \}\ \text{meth T::m}\ \{\ \mathbf{Q}\ \}\end{array}}$$

asumpt-axiom:

$$\mathbf{A}\vdash\ \mathbf{A}$$

asumpt–intro–rule:

$$\frac{\mathcal{A}\vdash\ \mathbf{A}}{\mathbf{A}_0\ ,\ \mathcal{A}\vdash\ \mathbf{A}}$$

assumpt–elim–rule:

$$\frac{\begin{array}{l}\mathcal{A}\vdash\ \mathbf{A}_0\\ \mathbf{A}_0\ ,\ \mathcal{A}\vdash\ \mathbf{A}\end{array}}{\mathcal{A}\vdash\ \mathbf{A}}$$

false–axiom:

$$\vdash\ \{\ \text{FALSE}\ \}\ \text{COMP}\ \{\ \text{FALSE}\ \}$$

inv–rule:

$$\frac{\mathcal{A}\vdash\ \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A}\vdash\ \{\ \mathbf{P}\wedge\mathbf{R}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\wedge\mathbf{R}\ \}}$$

where **R** is a $\Sigma$–formula.

conjunct–rule:

$$\frac{\begin{array}{l}\mathcal{A}\vdash\ \{\ \mathbf{P}_1\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1\ \}\\ \mathcal{A}\vdash\ \{\ \mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_2\ \}\end{array}}{\mathcal{A}\vdash\ \{\ \mathbf{P}_1\wedge\mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1\wedge\mathbf{Q}_2\ \}}$$

disjunct–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}_1\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1\ \} \qquad \mathcal{A} \vdash \{\ \mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_2\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}_1 \lor \mathbf{P}_2\ \}\ \text{COMP}\ \{\ \mathbf{Q}_1 \lor \mathbf{Q}_2\ \}}$$

strength–rule:

$$\frac{\mathbf{P}' \ \Rightarrow\ \mathbf{P} \qquad \mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}'\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}$$

weak–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \} \qquad \mathbf{Q} \ \Rightarrow\ \mathbf{Q}'}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}'\ \}}$$

subst–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[t/Z]\ \}\ \text{COMP}\ \{\ \mathbf{Q}[t/Z]\ \}}$$

where $Z$ is an arbitrary logical variable and $t$ a $\Sigma$–term.

all–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}[Y/Z]\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[Y/Z]\ \}\ \text{COMP}\ \{\ \forall Z : \mathbf{Q}\ \}}$$

ex–rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}[Y/Z]\ \}}{\mathcal{A} \vdash \{\ \exists Z : \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}[Y/Z]\ \}}$$

where $Z, Y$ are arbitrary, but distinct logical variables.

## A.3 Proof of Lemma 3.4

**Lemma 3.4** ($X$-Equivalence)

i) For any object $X$, $\equiv_X$ is an equivalence relation.

ii) $(\forall X : E \equiv_X E') \ \Rightarrow\ E = E'$

iii) $\neg reach(X, L, E) \ \Rightarrow\ E \equiv_X E\langle L := Y\rangle$

iv) $X \neq new(E, TID) \ \Rightarrow\ E \equiv_X E\langle TID\rangle$

v) $E \equiv_X E' \ \Rightarrow\ (reach(X, L, E) \ \Leftrightarrow\ reach(X, L, E'))$

The right hand side of the definition for $\equiv_X$ consists of two conjuncts. In the proofs of the 5 sublemmas, we refer to them as the *alive*–conjunct and the *reach*–conjunct.

**Proof of (i):** We have to show that for an arbitrary object $X$ the relation $\equiv_X$ is reflexive, transitive, and symmetric:

**Reflexive and Symmetric** Evident for the *alive*–conjunct. For the *reach*–conjunct it follows from lemma 3.2.(ix) .

**Transitive** Evident for the *alive*–conjunct. For the *reach*–conjunct we assume $\mathit{reach}(X, L, E)$ and derive $E(L) = E''(L)$:

$$\begin{array}{ll}
 & \mathit{reach}(X, L, E) \\
\Rightarrow & \quad [\![ \text{ from first premise and lemma 3.2.(ix) } ]\!] \\
 & E(L) = E'(L) \wedge \mathit{reach}(X, L, E') \\
\Rightarrow & \quad [\![ \text{ from second premise } ]\!] \\
 & E(L) = E'(L) \wedge E'(L) = E''(L) \\
\Rightarrow & \quad [\![ \text{ transitivity of equality } ]\!] \\
 & E(L) = E''(L)
\end{array}$$

**end of proof**

**Proof of (ii):** According to axiom env13, we have to show

$$(\forall X : \mathit{alive}(X, E) \Leftrightarrow \mathit{alive}(X, E')) \wedge (\forall L : E(L) = E'(L))$$

The first conjunct above follows from the *alive*–conjunct. To show the second conjunct above let $L$ be an arbitrary location. The premise of the sublemma yields $E \equiv_{\mathit{obj}(L)} E'$; as the premise of the the corresponding *reach*–conjunct, $\mathit{reach}(\mathit{obj}(L), L, E)$, is valid we conclude $E(L) = E'(L)$.

**end of proof**

**Proof of (iii):** The *alive*–conjunct follows directly from axiom env6. To show the *reach*–conjunct let $K$ be an arbitrary location. 1. Case: $\neg\mathit{reach}(X, K, E)$, i.e. the premise of the *reach*–conjunct does not hold. 2. Case: $\mathit{reach}(X, K, E)$. In this case the premise of the sublemma implies $K \neq L$ so that axiom env1 yields the needed equality $E(K) = E\langle L := Y\rangle(K)$.

**end of proof**

**Proof of (iv):** Concerning the *alive*–conjunct, axiom env7 and the premise yield:

$$\mathit{alive}(X, E\langle \mathit{TID}\rangle) \Leftrightarrow \mathit{alive}(X, E) \vee X = \mathit{new}(E, \mathit{TID}) \Leftrightarrow \mathit{alive}(X, E)$$

The *reach*–construct is obtained from env5.

**end of proof**

**Proof of (v):** From the *reach*–conjunct of $E \equiv_X E'$ by lemma 3.2.(ix) .

**end of proof**

## A.4 Proof of Lemma 3.13

To prove lemma 3.13, it suffices to derive

$$\vdash \{ \ \mathbf{P} \wedge \$ = E \wedge alive(X, \$) \wedge \bigwedge_{i=0}^{y} disj(X, \mathrm{p}_i, \$) \ \} \text{ meth T::m } \{ \ \$ \equiv_X E \ \}$$

based on the assumption that $\vdash \{ \ \mathbf{P} \ \}$ meth T::m $\{ \ \mathbf{Q} \ \}$ can be proved where $\mathrm{p}_i$, $i = 0, \ldots, y$, are the formal parameters of T::m. According to the rearranging technique of subsection 4.2.3.1 and the definition of $X$–equivalence this is equivalent to:

```
    $\{ \ \mathbf{P} \wedge \$ = E \wedge \bigwedge_{i=0}^{y} \mathrm{p}_i = P_i \ \}$
meth T::m
    $\{ \ alive(X, E) \wedge \bigwedge_{i=0}^{y} disj(X, P_i, E)$
        $\Rightarrow \ (alive(X, E) \Leftrightarrow alive(X, \$)) \wedge \forall L : reach(X, L, E) \ \Rightarrow \ E(L) = \$(L) \ \}$
```

The equivalence $alive(X, E) \Leftrightarrow alive(X, \$)$ is a direct consequence of the premise $alive(X, E)$ and lemma 3.10. Thus, it suffices to establish as postcondition:

$$alive(X, E) \wedge \bigwedge_{i=0}^{y} disj(X, P_i, E) \wedge reach(X, L, E) \ \Rightarrow \ E(L) = \$(L) \tag{A.1}$$

According to lemma 3.12, the assumed proof for $\vdash \{ \ \mathbf{P} \ \}$ meth T::m $\{ \ \mathbf{Q} \ \}$ enables to prove

```
    $\{ \ \mathbf{P} \wedge \$(L) = Y \wedge alive(obj(L), \$) \wedge \bigwedge_{i=0}^{y} \neg reach(\mathrm{p}_i, L, \$) \ \}$
meth T::m
    $\{ \ \$(L) = Y \wedge alive(obj(L), \$) \ \}$
```

Rearranging and simplification yields the following triple:

```
    $\{ \ \mathbf{P} \wedge \$ = E \wedge \bigwedge_{i=0}^{y} \mathrm{p}_i = P_i \ \}$
meth T::m
    $\{ \ alive(obj(L), E) \wedge \bigwedge_{i=0}^{y} \neg reach(P_i, L, E) \ \Rightarrow \ E(L) = \$(L) \ \}$
```

It remains to prove that the postcondition of this triple implies formula A.1, i.e.:

$$\begin{aligned} & alive(X, E) \wedge \textstyle\bigwedge_{i=0}^{y} disj(X, P_i, E) \wedge reach(X, L, E) \\ & \quad \Rightarrow \ alive(obj(L), E) \wedge \textstyle\bigwedge_{i=0}^{y} \neg reach(P_i, L, E) \end{aligned}$$

And this is a consequence of lemma 3.2.(viii) and the definition of predicate *disj*.

## A.5 Semantical Aspects of Subclassing

In subsection 4.1.1.3 it was said that subclassing often has a very operational semantics making it difficult to derive properties of the subclass from properties of the superclass. Here, we like to illustrate what we mean by operational semantics. Consider the following Java fragment that contains two classes. Class Intpair handles integer pairs and allows to sum the two components. Class Inttriple is a subclass of Intpair and extends it by a third integer component. Inttriple inherits the method sum from class Intpair and overwrites method add. The constructor methods are omitted:

```
class Intpair {
   int a;
   int b;
   int sum(){  return  this.add();  }
   int add(){  return  a+b;  }
   ...
}

class Inttriple extends Intpair {
   int c;
   int add(){  return super.sum() + c;  }
   ...
}
```

By the keyword `super` methods of the superclass can be called within the subclasses. This is the mechanism of Java to modify and adapt methods of superclasses (a typical example is the adaption of the event handler in the abstract windowing toolkit). From a naive point of view, method Inttriple::add could be read as follows: Call method Intpair::sum to sum up the first two components and add the third one. This view would assume that method Intpair::sum "keeps its meaning" in the subclass. Java explains subclassing and the problem of method selection in an operational way: A call of Inttriple::add causes a call of Intpair::sum. As the implicit parameter (denoted by `this` in Java) of this call is an Inttriple–object, the call of Intpair::sum causes a call of Inttriple::add which causes ... a StackOverflowError.

In addition to the operational selection rules illustrated by the example, the method selection mechanism depends on access rights. E.g. if method Intpair::add would have been declared private in the above example, the nonterminating mutual recursion would not occur. Such complex operational method selection mechanisms make it very difficult to derive semantical properties of a subclass from properties of the superclass.

## A.6 Implementation of Class PLIST

This section of the appendix presents the implementation of type PLIST as an example for a complete, non trivial class:

```
class  PLIST  is
   attr  arr: ARRAY
   attr  siz: INT
   attr  pos: INT

   meth  empty(): PLIST  is
      ilv: PLIST; av: ARRAY; iv: INT;
      av      :=  ARRAY::create(0) ;
      ilv     :=  new PLIST ;
      ilv.pos :=  0 ;
      ilv.arr :=  av ;
      ilv.siz :=  0 ;
      result  :=  ilv
   end

   meth  isempty(): BOOL  is
      iv: INT;
      iv     := self.siz ;
      result := iv.equ(0)
    end

    meth  first(): INT
       av: ARRAY; iv: INT; ov: OBJECT;
       iv     := self.siz  ;
       av     := self.arr  ;
       ov     := av.get(iv);
       result := (INT) ov
    end

    meth  rest(): PLIST
       ilv: PLIST; av: ARRAY; iv: INT;
       iv      :=  self.siz ;
       iv      :=  iv.add(-1) ;
       av      :=  self.arr ;
       ilv     :=  new PLIST ;
       ilv.arr :=  av ;
       ilv.pos :=  iv ;
       ilv.siz :=  iv ;
       result :=   ilv
    end

    meth  append( n: INT ): PLIST  is
       ilv: PLIST; av: ARRAY; iv: INT;
```

```
         iv :=  self.siz ;
         iv :=  iv.add(1) ;
         av :=  self.arr ;
         av :=  av.resize(iv) ;
         av.set(iv,n) ;
         ilv     :=  new PLIST ;
         ilv.pos :=  iv ;
         ilv.arr :=  av ;
         ilv.siz :=  iv ;
         result :=   ilv
      end

      meth  updfst( n: INT )  is
         av: ARRAY; iv: INT;
         iv :=  self.siz ;
         av :=  self.arr ;
         av.set(iv,n)
      end

      meth  init()  is
        iv: INT;
        iv := self.siz ;
        self.pos := iv
     end

     meth  isdef(): BOOL  is
        iv: INT;
        iv := self.pos ;
        result := 0.less(iv)
      end

      meth  next()  is
         iv: INT;
         iv :=  self.pos ;
         iv :=  iv.add(-1) ;
         self.pos := iv
      end

      meth  get(): INT  is
         av: ARRAY; iv: INT; ov; OBJECT;
         iv     :=  self.pos  ;
         av     :=  self.arr  ;
         ov     :=  av.get(iv);
         result :=  (INT) ov
      end
      meth  set( n: INT )  is
         av: ARRAY; iv: INT;
         iv :=  self.pos ;
         av :=  self.arr ;
         av.set(iv,n)
      end
   end
end
```

## A.7 Verification of a Sorting Method

In section 4.3.1 we gave a proof outline of method sort. This proof outline made use of the following property of the auxiliary method sorted_ins:

```
  meth  sorted_ins( n: INT; l: LIST ): LIST
     pre   aL(l,$) = a_sort(L) /\ n! = N /\ lng(aL(l,$)) < maxint
     post  aL(result,$) = a_sort(app(N,L))
```

The proof outline for the specification of method sorted_ins is similar to that of method sort. It uses the following properties of the function *a_sort*:

$$N \leq fst(a\_sort(L)) \;\Rightarrow\; app(N, a\_sort(L)) = a\_sort(app(N, L))$$

$$a\_sort(L) \neq empt \;\Rightarrow\; rst(a\_sort(L))) = a\_sort(rst(a\_sort(L)))$$

$$\begin{aligned} & N \geq fst(a\_sort(L)) \wedge L \neq empt \\ & \;\Rightarrow\; app(fst(a\_sort(L)), a\_sort(app(N, rst(a\_sort(L))))) = a\_sort(app(N, L)) \end{aligned}$$

The properties above are referenced by prop1–prop3 in the proof outline. Again we keep the invariants implicit:

```
     { $\text{n!} = N \wedge aL(\text{l},\$) = a\_sort(L) \wedge lng(aL(\text{l},\$)) < maxint$ }
  bv := l.isempty()  [[ functional property and no side–effects of isempty ]]
     { $\text{bv!} = isempt(aL(\text{l},\$)) \wedge \text{n!} = N \wedge aL(\text{l},\$) = a\_sort(L)$
       $\wedge\ lng(aL(\text{l},\$)) < maxint$ }
  if bv then
     { $isempt(aL(\text{l},\$)) \wedge \text{n!} = N \wedge aL(\text{l},\$) = a\_sort(L) \wedge lng(aL(\text{l},\$)) < maxint$ }
     ⇒ [[ $isempt(a\_sort(L)) \Rightarrow L = empt$ ]]
     { $L = empt \wedge \text{n!} = N \wedge aL(\text{l},\$) = L \wedge lng(aL(\text{l},\$)) < maxint$ }
  result := l.append(n);  [[ functional property and no side–effects of append ]]
     { $aL(\text{result},\$) = app(N, L) \wedge L = empt$ }
     ⇒ [[ $a\_sort(app(N, empt))$ ]]
     { $aL(\text{result},\$) = a\_sort(app(N, L))$ }
```

```
else
    $\{\ \neg isempt(aL(\mathrm{l},\$)) \wedge \mathrm{n!} = N$
      $\wedge\ aL(\mathrm{l},\$) = a\_sort(L) \wedge lng(aL(\mathrm{l},\$)) < maxint\ \}$
nv := l.first() ;
    $\{\ \mathrm{nv!} = fst(a\_sort(L)) \wedge \neg isempt(aL(\mathrm{l},\$)) \wedge \mathrm{n!} = N$
      $\wedge\ aL(\mathrm{l},\$) = a\_sort(L) \wedge lng(aL(\mathrm{l},\$)) < maxint\ \}$
bv := n.less(nv);
    $\{\ \mathrm{bv!} = (N < fst(a\_sort(L)))$
      $\wedge\ \mathrm{nv!} = fst(a\_sort(L)) \wedge \neg isempt(aL(\mathrm{l},\$)) \wedge \mathrm{n!} = N$
      $\wedge\ aL(\mathrm{l},\$) = a\_sort(L) \wedge lng(aL(\mathrm{l},\$)) < maxint\ \}$
if bv then
    $\{\ N < fst(a\_sort(L)) \wedge \mathrm{n!} = N$
      $\wedge\ aL(\mathrm{l},\$) = a\_sort(L) \wedge lng(aL(\mathrm{l},\$)) < maxint\ \}$
result := l.append(n)
    $\{\ aL(\mathrm{result},\$) = app(N, a\_sort(L)) \wedge N < fst(a\_sort(L))\ \}$
    $\Rightarrow\ [\![\ \mathrm{prop1}\ ]\!]$
    $\{\ aL(\mathrm{result},\$) = a\_sort(app(N,L))\ \}$
else
    $\{\ N \geq fst(a\_sort(L)) \wedge \mathrm{nv!} = fst(a\_sort(L)) \wedge \neg isempt(aL(\mathrm{l},\$))$
      $\wedge\ \mathrm{n!} = N \wedge aL(\mathrm{l},\$) = a\_sort(L) \wedge lng(aL(\mathrm{l},\$)) < maxint\ \}$
lv := l.rest();   $[\![\ lng(rst(a\_sort(L))) < lng(a\_sort(L))\ ]\!]$
    $\{\ N \geq fst(a\_sort(L)) \wedge \mathrm{nv!} = fst(a\_sort(L)) \wedge \neg isempt(a\_sort(L))$
      $\wedge\ \mathrm{n!} = N \wedge aL(\mathrm{lv},\$) = rst(a\_sort(L)) \wedge lng(aL(\mathrm{lv},\$)) < maxint\ \}$
    $\Rightarrow\ [\![\ \mathrm{prop2}\ ]\!]$
    $\{\ N \geq fst(a\_sort(L)) \wedge \mathrm{nv!} = fst(a\_sort(L)) \wedge L \neq empt \wedge \mathrm{n!} = N$
      $\wedge\ aL(\mathrm{lv},\$) = a\_sort(rst(a\_sort(L))) \wedge lng(aL(\mathrm{lv},\$)) < maxint\ \}$
lv := sorted_ins(n, lv);   $[\![$ here, assumption of recusion is used $]\!]$
                           $[\![$ no side–effect of sorted_ins is used without proof $]\!]$
    $\{\ N \geq fst(a\_sort(L)) \wedge L \neq empt \wedge \mathrm{nv!} = fst(a\_sort(L))$
      $\wedge\ aL(\mathrm{lv},\$) = a\_sort(app(N, rst(a\_sort(L)))) \wedge lng(aL(\mathrm{lv},\$)) < maxint\ \}$
result := lv.append( nv )
    $\{\ N \geq fst(a\_sort(L)) \wedge L \neq empt$
      $\wedge\ aL(\mathrm{result},\$) = app(fst(a\_sort(L)), a\_sort(app(N, rst(a\_sort(L)))))\ \}$
    $\Rightarrow\ [\![\ \mathrm{prop3}\ ]\!]$
    $\{\ aL(\mathrm{result},\$) = a\_sort(app(N,L))\ \}$
end
end
    $\{\ aL(\mathrm{result},\$) = a\_sort(app(N,L))\ \}$
end
```

# Bibliography

[Ada83] *Ada Programming Language*, 1983. ANSI/MIL-STD-1815A.

[AdB94] P. America and F. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6:269–316, 1994.

[Apt81] K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Trans. on Prog. Languages and Systems*, 3:431–483, 1981.

[BBB$^+$85] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickel, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP*. LNCS 183. Springer–Verlag, 1985.

[BFG$^+$93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The requirement and design specification language SPECTRUM: An informal introduction, version 1.0. Technical Report TUM-I9311/2, Technische Universität München, 1993.

[Bic95] K. Bichler. Specification of the C programming language. Master's thesis, Technische Universität München, 1995. (In German).

[Bij89] A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12:191–205, 1989.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta–Language*. LNCS 61. Springer–Verlag, 1978.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object–Oriented Programming Environments*. LNCS 562. Springer–Verlag, 1991.

[Bro96] M. Broy. Mathematical methods in system and software engineering. Working Material 8, International Summer School, Marktoberdorf, Germany, July–August 1996.

[CEW93] I. Classen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development*. AMAST Series in Computing 1. World Scientific, Singapore, 1993.

[CGR96] P. Chalin, P. Grogono, and T. Radhakrishnan. Identification of and solutions to shortcomings of LCL, a Larch/C interface specification language. In M.-C. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, LNCS 1051, pages 385–404. Springer–Verlag, 1996.

[CL94] Y. Cheon and G. T. Leavens. A quick overview of Larch/C++. *Journal of Object–Oriented Programming*, 7(6):39–49, October 1994.

[COR$^+$95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Technical report, SRI International, Menlo Park, CA 94025 USA, 1995.

[Cou90] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pages 841–993. North–Holland, Amsterdam, 1990.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison–Wesley, 1990.

[FJ92] L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design.* Cambridge Tracts in Theoretical Computer Science 35. Cambridge University Press, 1992.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, American Math. Soc., pages 19–32, 1967.

[GG89] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch prover. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, LNCS 355. Springer–Verlag, 1989.

[GH91] J. V. Guttag and J. J. Horning. A tutorial on Larch and LCL, a Larch/C interface language. In S. Prehn and W. J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, LNCS 552. Springer–Verlag, 1991.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer–Verlag, 1993.

[GMP90] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[Gor88] M. J. C. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall, 1988.

[Gri81] D. Gries. *The Science of Programming.* Springer–Verlag, 1981.

[GU91] T. Gergely and L. Úry. *First–Order Programming Theories.* Springer–Verlag, 1991.

[Gut75] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, Department of Computer Science, 1975. No. CSRG-59.

[GWM+92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-92-03, SRI International, 1992.

[Heh93] Eric C.R. Hehner. *A Practical Theory of Programming.* Texts and Monographs in Computer Science. Springer–Verlag, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic defintion of the programming language PASCAL. *Acta Informatica*, pages 335–355, 1973.

[JG96] G. Steele J. Gosling, B. Joy. *The Java Language Specification.* Addison–Wesley, Reading, MA, 1996.

[Kli93] P. Klint. A meta–environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Luc90] D. C. Luckham. *Programming with Specifications: An Introduction to Anna. A Language for Specifying Ada Programs.* Springer–Verlag, 1990.

[LW94] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[LZ75] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE-1:7–19, 1975.

[Mey88] Bertrand Meyer. *Object–Oriented Software Construction.* Prentice Hall, 1988.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object–Oriented Programming in the BETA Programming Language.* Addison–Wesley, 1993.

[Möl93] B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21:57–90, 1993.

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice Hall, 1994.

[OW97] M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In *The 24th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages.* ACM Press, 1997.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* Tr-75-251, Comp. Science Dept., Cornell University, 1975.

[Pau91] L. Paulson. *ML for the Working Programmer.* Cambridge Univ. Press, 1991.

[PH95] A. Poetzsch-Heffter. Interface specifications for program modules supporting selective updates and sharing and their use in correctness proofs. In G. Snelting, editor, *Softwaretechnik 95*, 1995.

[PN90] L. C. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, Computer Laboratory, 1990.

[Pol81] W. Polak. *Compiler Specification and Verification.* LNCS 124. Springer–Verlag, 1981.

[RT89] T. Reps and T. Teitelbaum. *The Syntheziser Generator Reference Manual.* Springer–Verlag, 1989. (3rd edition).

[SOM94] C. Szypersky, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. In J. Gutknecht, editor, *Programming Languages and System Architectures*, LNCS 782, pages 208–227. Springer–Verlag, 1994.

[Suz80] N. Suzuki, editor. *Automatic Verification of Programs with Complex Data Structures.* Garland Publishing, 1980.

[SVG79] Stanford Verification Group. Stanford Pascal Verifier user manual. Technical report, Stanford University, 79. Stanford Verification Group Report No. 11.

[Van93] M. T. Vandevoorde. Specifications can make programs run faster. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93*, LNCS 668. Springer–Verlag, 1993.

[Weg87] P. Wegner. Dimensions of object-based language design. In *Object Oriented Prog. Systems, Lang. and Applications.* ACM Press, 1987.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. MIT Press, 1993.

[Wir85] N. Wirth. *Programming in Modula–2.* Springer–Verlag, 1985.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pages 675–788. North–Holland, Amsterdam, 1990.

# Index

abstract class, 82
abstract interface, 74
abstract level, 17
abstract type, 34
abstraction–operator, 18
alive, 21
annotation
  of method, 48
  of statement, 48
antecedent, 50
application requirement, 24
assumption of sequent, 49

behaviour
  environmental, 18
  functional, 18
  sharing, 18

cast method, 76
class interface, 37
closed form of invariants, 98
closed type, 85
component, 48
concrete interface, 74
concrete type, 74
consequent of sequent, 49
constant, 9
contract, 11

declarative interface specification, 17
direct subtype, 31
direct supertype, 31
disjoint, 22
domain type, 37

environmental behaviour, 18
equivalence, 21
execution point, 47
exporting interfaces, 75

formula, 9
function signature, 9
functional behaviour, 18

genericity, 3

Hoare triple, 48

implicit parameter, 12
imported, 73
inherit, 82
instance of a rule, 59
interface
  abstract, 74
  concrete, 74
  private, 74
  public, 74, 75
interface invariant, 27
interface of a class, 37
interface of a module, 12
interface of a type, 12
interface specification
  declarative, 17
  operational, 14
interface specification of a type, 27
invariance property, 18, 21
invariant, 27

less alive, 55
local creation property, 111
local updating property, 70
location, 43
logic–based programming environment, 2
logical variable, 9

many–sorted signature, 9
method, 12
method annotation, 48

module, 12
  declared, 74
  interface of, 12
  predeclared, 74
  top, 74

name–closed, 28
normalform
  of method specifications, 23

object environment, 42
object held by location, 44
ObjEnv, 42
operational interface specification, 14

parameter
  implicit, 12
postcondition, 48
poststate, 47
precondition, 48
prestate, 47
prestate–operator, 19
prestate–term, 19
private, 74
procedure
  signature of, 12
program component, 48
programming environment
  logic–based, 2
public, 74

quantifier rules, 58

range type, 37
reachable, 52
refined partial correctness, 48
representation level, 17
requires clause, 24
rule, 50

satisfying an implementation, 15
semantic subtype relation, 32
semantic subtyping, 32
sequent, 49
set of interface specifications, 28
sharing behaviour, 18
sharing of objects, 24
sharing properties, 18
side–effects, 22
  absence of, 22
  free of, 22
signature, 9
  function, 9
signature of a procedure, 12
sort, 9
state, 47
statement annotation, 48
subclassing, 4, 82
subtype, 31
  direct, 31
subtype relation, 30
  semantic, 32
subtyping, 4
  semantic, 32
succedent, 50
supertype, 31
  direct, 31

term of a sort, 9
top module, 74
triple, 48
type
  abstract, 34
  closed, 85
  domain, 37
  interface of, 12
  interface specification of, 27
  range, 37
type correct, 63
type safe, 63

updating property, 70

variable
  logical, 9
variables of methods, 37

well–formed set of interfaces, 31
well–typed, 63

X–equivalence, 21