Interface Specifications for Program Modules Supporting Selective Updates and Sharing and their Use in Correctness Proofs

Arnd Poetzsch-Heffter
Institut für Informatik
Technische Universität
D-80290 München
poetzsch@informatik.tu-muenchen.de

Abstract

A new formal specification technique for imperative / object-oriented programs is presented, and its application to program verification is demonstrated. The technique supports the abstract specification of the functional as well as the sharing properties of program modules. In particular, procedures that make selective updates available to clients of modules can be specified in an implementation independent way.

Interface specifications of program modules can be used not only for documentation purposes, but as well for formal verification tasks. Therefore, the presented specification technique is designed so that specifications can be directly applied as logical rules in Hoare-style correctness proofs. As an example, we prove the correctness of a sort procedure that exploits selective updates. Finally, implementations of interface specifications are discussed.

Keywords: Interface specification, specification of imperative/object-oriented programs, program verification

1 Introduction

An interface specification of a software module can be considered as a contract between the client of the module and the implementor. The interface specification formally describes all aspects of the module that are relevant for the client. All other aspects should be hidden. Interface specifications are given in so-called interface specification languages that link the operational notions of programs to general program independent specifications. A lot of theoretical and practical work has been invested in the last ten years to develop interface specification languages and tool support for them and to fill the gap between general (e.g. algebraic) specification languages and realistic programming languages (cf. [GH93]). In this paper, we present a method that allows to extend and improve the existing approaches in two aspects:

1. Sharing properties: Sharing of data structures and selective updates are essential programming techniques to enable efficient implementations of composed complex data structures (most module libraries for realistic imperative/object-oriented languages make use of these techniques). Consequently, at least some of the sharing properties of program modules have to be made visible to the client so that he can use updating operations and avoid copying whenever possible. We show how sharing properties can be specified in a formal, implementation independent way.

2. Proving applications correct: An interface specification should not only be a formal description of the module properties, but should provide the basis to prove client programs correct. We show how our interface specifications can be used as rules in correctness proofs.

The basic idea of our approach is to specify the functional behaviour of a module by using abstract data types and to specify the sharing behaviour by abstract sharing functions or predicates; e.g. a predicate that tells whether two object structures are disjoint, i.e. do not share storage. From a top-down software engineering point of view, a module is an implementation of an abstract data type such that each function of the ADT is implemented by at least one procedure in the module. In addition to that, the module can provide other procedures, e.g. procedures with selective updates, to allow for more efficiency, and a copy procedure.

The work is part of a project that aims at the systematic construction of a new generation of language-specific programming environments that provide — in addition to the classical features — specification and proof support. As our approach suits well with the encapsulation and inheritance principles of object-oriented software development, we illustrate it by using an object-oriented programming language. But

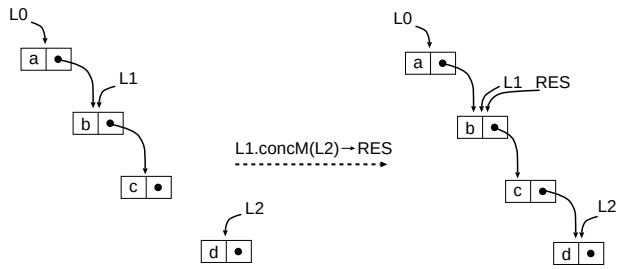
the method can be applied to other imperative programming languages as well.

Overview Section 2 introduces our interface specification method, in particular the way we express sharing properties, and discusses related work. Section 3 explains how our interface specifications can be used in correctness proofs of client programs. Section 4 sketches different implementations that satisfy the example interface specification of section 2, and discusses implementation correctness.

2 Specifying Data Type Implementations with Sharing

2.1 The Goal

If we implement abstract data types by object-oriented programs, a "structured" value, e.g. a list, is usually represented by an object structure, e.g. by a collection of linked objects. Many operations can be implemented much more efficiently, if we allow them to modify existing object structures; but such modifications can produce side-effects and destroy referential transparency. Consider a concatenation method concM on linked lists applied as L1.concM(L2): By updating the last list element of its first parameter L1, it can produce an object structure representing the result list; but this modification produces side-effects on L1 and all lists shared with it (here L0).



Certainly for many simple applications, implementations that guarantee referential transparency are sufficiently efficient. But in a lot of cases, updating operations are indispensable. E.g. if graphs are implemented by two dimensional matrices, insertion and deletion of edges is very efficient as long as we implement them as updates. Or consider a module implementing a file system. In deed, all software libraries for imperative/object-oriented programming languages we know of provide updating and copy operations. Thereby, they transfer some of the responsibility of managing shared object structures to their clients. In order to meet this responsibility, at least some aspects of how sharing is done has to be visible to the clients. Consequently, the central goal is to provide interface specification techniques that allow to specify sharing properties in an abstract way, i.e. without refering to the implementation. There are several reasons for using such interface specifications instead of refering to the implementation directly:

- The client should not rely on implementation details so that later changes to the module do not effect client programs.
- In general, specifications are shorter and better to read.
- Interface specifications can be used in verification:
  - The client can use them to prove properties of procedures that are implemented based on the corresponding modules.
  - The implementor can prove that the implementation satisfies the interface specification.

To illustrate our approach to interface specification and the corresponding verification method, we will use the following class CList that implements the abstract data type LIST with functions top, rest, app, conc, empty, and isempty (cf. appendix for the function signatures):

```
class interface  CList
    method  ::emptyM(): CList
    method  isemptyM(): Bool
    method  topM(): Elem
    method  restM(): CList
    method  appM( E: Elem ): CList
    method  concM( L: CList ): CList
    method  copyM(): CList
    method  updtopM( E: Elem ): CList
end
```

To keep things simple here, we consider classes as modules (like in several object-oriented programming languages, e.g. Sather). For each function of the abstract data type LIST, the class CList provides one method. All methods except emptyM take a CList object as implicit first parameter, the so-called SELF parameter; emptyM is a class method and called by CList::emptyM(). In order to support efficient and space saving implementations the method updtopM simply "updates the first element" of a given list L, thereby producing side-effects on all lists shared with L. Similar, concM may modify its SELF parameter possibly producing side-effects on lists shared with SELF.

2.2 The Approach

In this paper we consider only interface specifications for procedures/methods. However in our whole framework, we work with other specification constructs, in particular invariants of classes, as well. The interface specification of a method consists of three parts: a precondition PRE, an invariant condition INV, and

an ensures clause ENS. PRE and ENS are sorted first-order formulas, invariant conditions are explained below.

Methods that handle shared object structures representing abstract values have two aspects: a *functional* aspect, explaining the behaviour of the method in terms of the abstract values; and a *sharing* aspect, explaining the side-effects and modifications done to object structures.

The basic idea of the presented method is to specify the functional behaviour by an *abstraction function* and the sharing behaviour by *sharing functions* or predicates. (How such functions can be specified or derived is discussed in section 4.) The abstraction function takes an object OBJ in an execution state and yields the abstract value of the ADT that corresponds to the object structure reachable from OBJ. We illustrate the specification technique by the above list example. The abstraction function abs yields for an object of class CList the corresponding value of the abstract data type LIST. Sharing of list representations is expressed by the predicates

isdisj : CList x CList → Bool
isdirpart : CList x CList → Bool

Informally, isdisj(L1,L2) yields true, iff the object structures corresponding to L1 and L2 do not share storage; isdirpart(L1,L2) yields true, iff the object structure corresponding to L1 is a direct part of that corresponding to L2 and abs(L1) = rest(abs(L2)). Whereas an intuitive understanding of abstraction and sharing functions is certainly helpful in practice, it is not necessary from a formal point of view, as their relevant properties are axiomatized within the interface specification.

We start the explanation of interface specifications by having a look at the specifications for the methods isemptyM and emptyM:

SLF.isemptyM() → RES
pre abs(SLF) = $\overline{\text{SLF}}$
inv true
ens RES = isempty($\overline{\text{SLF}}$)

CList::emptyM() → RES
inv true
ens abs(RES) = empty

The specification of isemptyM is read as follows: Let SLF denote the parameter and RES the result of a call to the method isemptyM; if the abstraction of the object structure referenced by SLF represents the list $\overline{\text{SLF}}$ at the beginning of a call to isemptyM, then the result of that call equals isempty($\overline{\text{SLF}}$). The identifiers SLF, $\overline{\text{SLF}}$, L, and RES are free logical variables; thus, consistent renaming does not change the meaning of the specification. By convention, we will denote logical variables of sort List by overlined identifiers.

The precondition and ensures clause specify that the method isemptyM implements the predicate isempty; i.e. they formulate that abs is a homomorphism w.r.t. isemptyM. The invariant condition which is explained later guarantees that isemptyM has no side-effects. For the method emptyM the precondition is true and can therefore be omitted. Already with this simple example, we can see two ways how the classes/types of the programming language are linked to the sorts of the abstract data type: Whereas the class CList is linked by the abstraction function abs to sort List, the type Bool is taken to be identical to the sort Bool; that is why we can write RES = isempty($\overline{\text{SLF}}$). We assume that the same holds for the element type Elem of lists.

In the following, we present the interface specifications for topM and restM. Predicate applications of the form "predname[...]" are used as abbreviations for "predname(...)=true":

SLF.topM() → RES
pre ¬ isempty[abs(SLF)] ∧ abs(SLF) = $\overline{\text{SLF}}$
inv true
ens RES = top($\overline{\text{SLF}}$)

SLF.restM() → RES
pre ¬ isempty[abs(SLF)] ∧ abs(SLF) = $\overline{\text{SLF}}$
inv true
ens abs(RES) = rest($\overline{\text{SLF}}$)
∧ isdirpart[RES,SLF]

Looking closer to the specification of method restM, it has three aspects: the functional aspect, the invariant aspect, and the sharing aspect that guarantees that the object structure referenced by the result is a direct part of the object structure referenced by SLF. In particular, restM does not copy the list representation. As we want to use specifications later on as proof rules, we try to keep them as modular as possible; i.e. we prefer to give the specification of restM in three seperate parts:

SLF.restM() → RES
pre ¬ isempty[abs(SLF)] ∧ abs(SLF) = $\overline{\text{SLF}}$
ens abs(RES) = rest($\overline{\text{SLF}}$)

SLF.restM() → RES
pre ¬ isempty[abs(SLF)]
inv true

SLF.restM() → RES
pre ¬ isempty[abs(SLF)]
ens isdirpart[RES,SLF]

The specification parts are called (specification) axioms. The seperation into axioms is not only more convenient for proofs, but in general a stronger formulation, because (a) it tells exactly which preconditions are needed to establish which part of the ensures clause and (b) the composed version can be obtained by an obvious conjunction law.

Before we explain the invariant condition, we give

the specifications for copyM and concM (specifications for the remaining methods can be found in the appendix):

SLF.copyM() → RES
pre abs(SLF) = $\overline{\text{SLF}}$
ens abs(RES) = $\overline{\text{SLF}}$

SLF.copyM() → RES
inv true

SLF.copyM() → RES
ens isdisj[L, RES]

L1.concM(L2) → RES
pre isdisj[L1,L2] ∧ abs(L1) = $\overline{\text{L1}}$ ∧ abs(L2) = $\overline{\text{L2}}$
ens abs(RES) = conc($\overline{\text{L1}}$,$\overline{\text{L2}}$)

L1.concM(L2) → RES
pre isdisj[L1,L2]
inv L: isdisj[L,L1] ∧ isdisj[L,L2]

L1.concM(L2) → RES
pre isdisj[L1,L2] ∧ isdisj[L,L1] ∧ isdisj[L,L2]
ens isdisj[L,RES]

As for restM, there are three axioms for copyM and for concM. They specify the functional, invariant, and sharing behaviour of the three methods. For later reference, they are denoted by *restFUNCT*, *restINV*, *restSHARE*, *copyFUNCT*, *copyINV*, *copySHARE*, *concFUNCT*, *concINV*, and *concSHARE*.

Whereas the use of preconditions and ensures clauses (postconditions) is well known and widely used in the literature, our form of invariant condition deserves explanation. As the name suggests, invariant conditions specify what properties are guaranteed to be left invariant under a method execution. Of course, a huge class of properties is left invariant under executions of methods of class CList; in particular, all those properties that do not refer to object structures of CList at all. In order to characterize a subclass of the invariant properties that is sufficient for our purposes, we consider all properties that are expressible by propositions, i.e. quantifier free formulas. (Notice that this is sufficiently powerful to capture the assertion mechanisms of existing programming languages like e.g. in Eiffel). Obviously, in general a proposition is not invariant under the execution of a method; e.g. abs(L1) = $\overline{\text{L1}}$ is not invariant for L1.concM(L2) → RES, if the concM implementation works by selective update as illustrated in section 2.1: After execution of such implemented concM, we have abs(RES) = abs(L1) = conc($\overline{\text{L1}}$,$\overline{\text{L2}}$) which is in general unequal to $\overline{\text{L1}}$.

In many cases, propositions are only invariant for a method body, if their free variables satisfy special conditions. These conditions are formulated by the invariant clauses. E.g. let PROP[$Y_1, \ldots, Y_n$] be a proposition and $Y_1, \ldots, Y_k$ be the variables of sort CList (we assume here that sort information can be always derived; in particular, we derive that the free variable L in *concINV* is of sort CList). Then, the specification axiom *concINV* states that a proposition PROP[$Y_1, \ldots, Y_n$] is invariant for concM as long as the $Y_1, \ldots, Y_k$ are disjoint from the arguments of concM; i.e. we consider *concINV* as an abbreviation for

L1. concM(L2) → RES
pre isdisj[L1,L2] ∧ PROP[$Y_1, \ldots, Y_n$]
∧ $\bigwedge_{j=1}^{k}$(isdisj[Y_j,L1] ∧ isdisj[Y_j,L2])
ens PROP[$Y_1, \ldots, Y_n$]

In particular, we can derive axioms like:

L1. concM(L2) → RES
pre isdisj[L1,L2] ∧ abs(L0) = $\overline{\text{L0}}$
∧ isdisj[L0,L1] ∧ isdisj[L0,L2]
ens abs(L0) = $\overline{\text{L0}}$

More generally, an invariant clause is a list of pairs of the form X: PROP[X,..] where X is a variable and PROP a proposition. If the list is empty, we denote it by "true".
As for similar specification methods, quite a lot of syntactical abbreviations can be applied to make the interface specifications more readable and compact (see e.g. [Jon91] for a discussion). E.g. introducing the notion of a general precondition of a method M, i.e. a condition that is part of all preconditions in axioms for M (e.g. isdisj[L1,L2] for concM), we can drop the functional axioms as they have always the same form[1]. Here we tried to explain the specification method using a fairly basic notation simply to keep the focus on the central aspects and to avoid a mixture with syntactical issues.

In addition to the method specifications, the interface specification may contain first-order axioms characterizing the used functions and predicates. In the proofs of the section 3 for example, we will need the following properties:

isdisjCOMM: isdisj(L1,L2) = isdisj(L2,L1)

isdirpartABS: isdirpart[L1, L2]
→ abs(L2) = app(top(abs(L2)), abs(L1))

2.3 Related Work and Discussion

The presented work lives in the triangle of (a) specification and (b) verification of imperative/object-oriented programs and (c) their use for implementing abstract data types. Our approach follows the two-tiered specification style of the Larch family of languages ([GH93]); i.e. there is an algebraic specification language that is independent from programming languages; and for each programming language, there is a specific interface language. The interface language bridges the gap between the state-less world of al-

[1] Assuming that the method name indicates which function of the abstract data type it implements.

gebraic specifications and the state-based behaviour of imperative programming languages. Our interface specification method differs in two aspects from interface specifications such as LCL [GH91], the interface language for C:

- Larch interface languages essentially support two specification styles: concrete and abstract types (Larch/C++ [CL94] provides additional techniques that allow to write specifications similar to ours, but on a less abstract level). Concrete types are types provided by the programming languages that are directly modelled without an abstraction layer between specification and implementation; in our example, the type Elem was treated as a concrete type. If a type T is declared to be abstract, identifiers of this type occurring in interface specifications are assumed to have values of the abstract data type that corresponds to T; i.e. what we make explicit by the abstraction function is left implicit. The implementation of abstract types has to guarantee referential transparency in Larch. Consequently, the interface of implementations with selective updates cannot be specified. In this respect, our method supports a bigger class of implementations.

- Instead of our invariant conditions, Larch interface languages provide so-called *modifies clauses*. A modifies clause names all variables that a procedure is allowed to modify. In deed for global variables, it is usually simpler to describe directly what is modified than to give an invariant. On the other hand for object structures or (recursive) pointer data structures, this approach has two disadvantages: (a) Without auxiliary functions it cannot be specified which variables are modified by a procedure (e.g. in our conc example, a function is needed that yields the instance variable tail of the last list object of L1). (b) With modifies clauses it is much harder to abstract from implementation details, as modifies clauses need to refer to variables, i.e. to the concrete representation (cf. section 4). Another disadvantage of modifies clauses compared to invariants is that reasoning about the correctness of client programs or implementations becomes more difficult. But it should be pointed out here that the primary concern of the family of Larch interface languages is to enable and support *specification* for existing programming languages.

The COLD approach to interface specifications (cf. [Jon89]) is similar to the Larch approach in that it also uses the notion of modification in interfaces. But, COLD is not primarily interested in specifying real programs as it is our goal and the goal of Larch interface languages. COLD is mainly developed to specify state-based systems (in particular, the variables in the modifies clauses need not correspond to variables in a real implementation, but are used as specification concepts). In COLD, reasoning is supported by a version of dynamic logic.

Another aspect of our work is the implementation of abstract data types by imperative/object-oriented programs. E.g. R. Breu ([Bre91]) developed a framework for formally relating object-oriented specifications of abstract data types to object-oriented programs. In her method "the implementation of a method is correct w.r.t. an operation if it produces the proper result, independent of side-effects. ... However, the information provided by an abstraction function is not sufficient to reason about programs which depend on side-effects" (cf. p. 175f). That is exactly the point where our method can help by refining the implementation relation.

3 Using Interface Specifications in Correctness Proofs

As pointed out in the discussion of related work above, interface specifications will and have to play an important role for realistic program verification. Therefore, our interface specifications are designed in such a way that they can be directly used in Hoare-style correctness proofs ([Hoa69]). This section explains how properties of client programs can be proved. By *client programs* we mean programs that use object structures only through module/class interfaces. In particular, client programs cannot directly update object attributes. The correctness of implementations is discussed in section 4.

3.1 Introduction to Correctness Proofs

As in Hoare-logic, theorems in our logic are triples consisting of a precondition, a program fragment, and an ensures clause (postcondition). In particular, a specification axiom having no invariant condition can be considered as a Hoare-triple; and in deed, we use Hoare's notation in proofs, i.e. in addition to the form used above, we write e.g. for *copyFUNCT*:

$$\begin{array}{l} \quad \mathrm{abs(SLF)} = \overline{\mathrm{SLF}} \\ \{\ \mathrm{SLF.copyM()} \rightarrow \mathrm{RES}\ \} \\ \quad \mathrm{abs(RES)} = \overline{\mathrm{SLF}} \end{array}$$

In contrast to classical Hoare-logic, we have to deal with side-effects in expressions. This is done by decomposing expressions according to the evaluation order[2] and using (fresh) variables for intermediate results. These result variables may only be used in the ensures clause of the corresponding Hoare-triple. As

[2] If the evaluation order is not fixed by the programming lanuguage, the goal Hoare-triple has to be proven for the set of all legal evaluation orders; in practice this set can be reduced by quotient techniques.

an example, we use the interface specification of section 2 to prove a simple theorem about concM and copyM:

Assuming that two CList objects L1, L2 are disjoint, the concatenation of the result of L1.concM(L2.copyM()) with L2 yields an object structure the abstraction of which equals conc(conc($\overline{L1}$, $\overline{L2}$), $\overline{L2}$). Without the call to copyM, the theorem is in general not true, e.g. for the linked list implementation of section 2.1 the resulting object structure would contain a cycle.

isdisj[L1, L2] $\wedge$ abs(L1) = $\overline{L1}$ $\wedge$ abs(L2) = $\overline{L2}$
{ L1.concM(L2.copyM()). concM(L2) $\rightarrow$ RES }
abs(RES) = conc(conc($\overline{L1}$, $\overline{L2}$), $\overline{L2}$)

Proof: The decomposition of the expression yields three parts. For each of these parts we prove a Hoare-triple such that the ensures clause of the first triple equals the precondition of the second, and the ensures clause of the second equals the precondition of the third:

Lemma 1:
isdisj[L1,L2] $\wedge$ abs(L1) = $\overline{L1}$ $\wedge$ abs(L2) = $\overline{L2}$
{ L2.copyM() $\rightarrow$ RES1 }
isdisj[L1,L2] $\wedge$ abs(L1) = $\overline{L1}$ $\wedge$ abs(L2) = $\overline{L2}$
$\wedge$ isdisj[L1,RES1] $\wedge$ isdisj[L2,RES1]
$\wedge$ abs(RES1) = $\overline{L2}$

Lemma 2:
isdisj[L1,L2] $\wedge$ abs(L1) = $\overline{L1}$ $\wedge$ abs(L2) = $\overline{L2}$
$\wedge$ isdisj[L1,RES1] $\wedge$ isdisj[L2,RES1]
$\wedge$ abs(RES1) = $\overline{L2}$
{ L1.concM(RES1) $\rightarrow$ RES2 }
isdisj[RES2,L2] $\wedge$ abs(RES2) = conc($\overline{L1}$, $\overline{L2}$)
$\wedge$ abs(L2) = $\overline{L2}$

Lemma 3:
isdisj[RES2,L2] $\wedge$ abs(RES2) = conc($\overline{L1}$, $\overline{L2}$)
$\wedge$ abs(L2) = $\overline{L2}$
{ RES2. concM(L2) $\rightarrow$ RES }
abs(RES) = conc(conc($\overline{L1}$, $\overline{L2}$), $\overline{L2}$)

To prove lemma 1, we use four triples: (a) derived from *copyINV*, (b) a renamed version of *copySHARE*, (c) a renamed version of *copySHARE*, and (d) a renamed version of *copyFUNCT*:

isdisj[L1,L2] $\wedge$ abs(L1) = $\overline{L1}$ $\wedge$ abs(L2) = $\overline{L2}$
{ L2.copyM() $\rightarrow$ RES1 }
isdisj[L1,L2] $\wedge$ abs(L1) = $\overline{L1}$ $\wedge$ abs(L2) = $\overline{L2}$

true
{ L2.copyM() $\rightarrow$ RES1 }
isdisj[L1,RES1]

true
{ L2.copyM() $\rightarrow$ RES1 }
isdisj[L2,RES1]

abs(L2) = $\overline{L2}$
{ L2.copyM() $\rightarrow$ RES1 }
abs(RES1) = $\overline{L2}$

The conjunctive composition of these four triples yields lemma 1. To prove lemma 2, we use renamed instances of *concSHARE*, *concFUNCT*, and of the triple that we derived at the end of section 2.2 from *concINV*:

isdisj[L1,RES1] $\wedge$ isdisj[L2,L1] $\wedge$ isdisj[L2,RES1]
{ L1.concM(RES1) $\rightarrow$ RES2 }
isdisj[L2,RES2]

isdisj[L1,RES1] $\wedge$ abs(L1) = $\overline{L1}$ $\wedge$ abs(RES1) = $\overline{L2}$
{ L1.concM(RES1) $\rightarrow$ RES2 }
abs(RES2) = conc($\overline{L1}$, $\overline{L2}$)

isdisj[L1,RES1] $\wedge$ abs(L2) = $\overline{L2}$
$\wedge$ isdisj[L2,L1] $\wedge$ isdisj[L2,RES1]
{ L1.concM(RES1) $\rightarrow$ RES2 }
abs(L2) = $\overline{L2}$

Taking the conjunctive composition of the above triples and exploiting the commutativity of $\wedge$ and isdisj, we get lemma 2. Finally, lemma 3 is an instance of *concFUNCT*.

3.2 Correctness of Procedures Exploiting Sharing and Selective Updates

In this section, we provide a central part of a proof showing the partial correctness of a sorting procedure sortM. sortM is based on class CList and gains a lot from exploiting selective updates and sharing. It sorts a list represented by an object structure simply by exchanging the list elements, i.e. without touching the object structure itself. The main aspect of this section is to illustrate that client programs using complex sharing properties and selective updates can be handled by our approach to specification and verification. It goes without saying that the following proof is independent from CList implementations, i.e. it holds for all CList implementations satisfying the interface specification (in particular, it holds for the implementations sketched in section 4).

The following method sortM sorts a list L by first sorting the rest of L and then calling the auxiliary method sorted_insM that inserts the first element of L into the sorted rest:

```
method sorted_insM(): CList
  if   SELF.restM().isemptyM() then  SELF
  elsf SELF.topM() < SELF.restM().topM()
    then  SELF
    else
      Elem ELEMVAR :=  SELF.topM();
      SELF. updtopM( SELF.restM().topM() );
      SELF. restM().
         updtopM( ELEMVAR ). sorted_insM();
      SELF
end
```

```
method sortM(): CList
  if  SELF.isemptyM()  then  SELF
  else
     SELF.restM().sortM();
     SELF.sorted_insM()
end
```

There are several ways to formulate the statement that sortM is partially correct. We assume here that the abstract data type LIST provides a function sort. Using sort, we can formulate the correctness theorem *sortPCORR* as follows:

$$\begin{array}{l} \quad \mathrm{abs(SLF)} = \overline{\mathrm{SLF}} \\ \{\ \mathrm{SLF.sortM()} \rightarrow \mathrm{RES}\ \} \\ \quad \mathrm{abs(RES)} = \mathrm{sort}(\overline{\mathrm{SLF}}) \wedge \mathrm{RES} = \mathrm{SLF} \end{array}$$

The conjunct "RES = SLF" is a necessary embedding, as we will see. We have here neither the room nor did we provide the logical means[3] to formally prove the above theorem. Rather, we focus on a central part of the proof by which we aim to illustrate how functional and sharing aspects are intertwined.

For partial correctness, it suffices to show that *sortPCORR* can be proven for the body of sortM under the assumption that it holds for all calls to sortM in the body. The essential part of the proof is to show the following for the else-branch of sortM:

$$\begin{array}{l} \quad \neg\ \mathrm{isempty[abs(SLF)]} \wedge \mathrm{abs(SLF)} = \overline{\mathrm{SLF}} \\ \{\ \mathrm{SLF.restM().sortM();\ SLF.sorted\_insM()} \rightarrow \mathrm{RES}\} \\ \quad \mathrm{abs(RES)} = \mathrm{sort}(\ \overline{\mathrm{SLF}}\ ) \wedge \mathrm{RES} = \mathrm{SLF} \end{array}$$

The rest of the proof is obtained by the invariant property of isempty and the rule for conditional expressions (not shown). The proof for the else-branch is given as a derivation chain (the derivation steps are annotated with arguments and lemmas that are given below):

$$\begin{array}{l} \quad \neg\ \mathrm{isempty[abs(SLF)]} \wedge \mathrm{abs(SLF)} = \overline{\mathrm{S}} \\ \{\ \mathrm{SLF.restM()} \rightarrow \mathrm{L}\ \} \\ \quad \text{using } \textit{restFUNCT, restSHARE, restINV} \\ \quad \mathrm{abs(L)} = \mathrm{rest}(\overline{\mathrm{S}}) \wedge \mathrm{isdirpart[L,SLF]} \\ \qquad \wedge\ \mathrm{abs(SLF)} = \overline{\mathrm{S}} \\ \Rightarrow \qquad \text{using "A = B} \rightarrow \mathrm{top(A)} = \mathrm{top(B)}\text{"} \\ \quad \mathrm{abs(L)} = \mathrm{rest}(\overline{\mathrm{S}}) \wedge \mathrm{isdirpart[L,SLF]} \\ \qquad \wedge\ \mathrm{top(abs(SLF))} = \mathrm{top}(\overline{\mathrm{S}}) \\ \{\ \mathrm{L.sortM()} \rightarrow \mathrm{RES}\ \} \\ \quad \text{using } \textit{sortPCORR, sortISDIRP\_INV, sortTOP\_INV} \\ \quad \mathrm{abs(RES)} = \mathrm{sort(rest}(\overline{\mathrm{S}})) \wedge \mathrm{RES} = \mathrm{L} \\ \qquad \wedge\ \mathrm{isdirpart[L,SLF]} \wedge \mathrm{top(abs(SLF))} = \mathrm{top}(\overline{\mathrm{S}}) \\ \Rightarrow \qquad \text{using "RES = L"} \\ \quad \mathrm{abs(L)} = \mathrm{sort(rest}(\overline{\mathrm{S}})) \wedge \mathrm{isdirpart[L,SLF]} \\ \qquad \wedge\ \mathrm{top(abs(SLF))} = \mathrm{top}(\overline{\mathrm{S}}) \\ \Rightarrow \qquad \text{using } \textit{isdirpartABS} \\ \quad \mathrm{abs(L)} = \mathrm{sort(rest}(\overline{\mathrm{S}})) \\ \qquad \wedge\ \mathrm{abs(SLF)} = \mathrm{app(top(abs(SLF)),\ abs(L)}\ ) \\ \qquad \wedge\ \mathrm{top(abs(SLF))} = \mathrm{top}(\overline{\mathrm{S}}) \\ \Rightarrow \qquad \text{using "}\neg\ \mathrm{isempty[app(E,L)]}\text{"} \\ \quad \neg\ \mathrm{isempty[abs(SLF)]} \\ \qquad \wedge\ \mathrm{abs(SLF)} = \mathrm{app(top}(\overline{\mathrm{S}}),\ \mathrm{sort(rest}(\overline{\mathrm{S}}))) \\ \Rightarrow \qquad \text{using } \textit{restsortPROP} \\ \quad \neg\ \mathrm{isempty[abs(SLF)]} \\ \qquad \wedge\ \mathrm{abs(SLF)} = \mathrm{app(}\ \mathrm{top}(\overline{\mathrm{S}}),\ \mathrm{sort(rest}(\overline{\mathrm{S}}))\ ) \\ \qquad \wedge\ \mathrm{rest(}\ \mathrm{app(top}(\overline{\mathrm{S}}),\mathrm{sort(rest}(\overline{\mathrm{S}})))\ ) \\ \qquad\quad = \mathrm{sort(rest(}\ \mathrm{app(top}(\overline{\mathrm{S}}),\mathrm{sort(rest}(\overline{\mathrm{S}})))\ )) \\ \{\ \mathrm{SLF.sorted\_insM()} \rightarrow \mathrm{RES}\ \} \\ \quad \text{using } \textit{sorted\_insLEMMA} \\ \quad \mathrm{abs(RES)} = \mathrm{sort(}\ \mathrm{app(top}(\overline{\mathrm{S}}),\mathrm{sort(rest}(\overline{\mathrm{S}})))\ ) \\ \qquad \wedge\ \mathrm{RES} = \mathrm{SLF} \\ \Rightarrow \qquad \text{using } \textit{sortIDEMPOT} \\ \quad \mathrm{abs(RES)} = \mathrm{sort(}\ \overline{\mathrm{S}}\ ) \wedge \mathrm{RES} = \mathrm{SLF} \end{array}$$

Besides the assumption *sortPCORR* for the recursive call and the axiom *isdirpartABS* that comes with the interface specification (cf. end of section 2.2), the above proof uses two equalities[4] (shown by laws of the abstract data type LIST) and three lemmas:

restsortPROP:
$$\begin{array}{l} \quad \mathrm{rest(}\ \mathrm{app(top}(\overline{\mathrm{S}}),\mathrm{sort(rest}(\overline{\mathrm{S}})))\ ) = \mathrm{sort(rest}(\overline{\mathrm{S}})) \\ \quad = \mathrm{sort(}\ \mathrm{sort(rest}(\overline{\mathrm{S}}))\ ) \\ \quad = \mathrm{sort(}\ \mathrm{rest(}\ \mathrm{app(}\ \mathrm{top}(\overline{\mathrm{S}}),\ \mathrm{sort(rest}(\overline{\mathrm{S}}))\ )\ )) \end{array}$$

sortIDEMPOT:
$$\begin{array}{l} \quad \mathrm{sort(}\ \mathrm{app(top}(\overline{\mathrm{S}}),\mathrm{sort(rest}(\overline{\mathrm{S}})))\ ) \\ \quad = \mathrm{sort(}\ \mathrm{app(}\ \mathrm{top}(\overline{\mathrm{S}}),\ \mathrm{rest}(\overline{\mathrm{S}})\ )\ ) = \mathrm{sort(}\ \overline{\mathrm{S}}\ ) \end{array}$$

sortISDIRP_INV:
$$\begin{array}{l} \quad \mathrm{isdirpart[L1,L2]} \\ \{\ \mathrm{SLF.sortM()} \rightarrow \mathrm{RES}\ \} \\ \quad \mathrm{isdirpart[L1,L2]} \end{array}$$

sortTOP_INV:
$$\begin{array}{l} \quad \mathrm{isdirpart[SLF,L]} \wedge \mathrm{top(abs(L))} = \mathrm{E} \\ \{\ \mathrm{SLF.sortM()} \rightarrow \mathrm{RES}\ \} \\ \quad \mathrm{top(abs(L))} = \mathrm{E} \end{array}$$

sorted_insLEMMA:
$$\begin{array}{l} \quad \neg\ \mathrm{isempty[}\ \mathrm{abs(SLF)}\ \mathrm{]} \wedge \mathrm{abs(SLF)} = \overline{\mathrm{S}} \\ \qquad \wedge\ \mathrm{rest}(\overline{\mathrm{S}}) = \mathrm{sort(rest}(\overline{\mathrm{S}})) \\ \{\ \mathrm{SLF.sorted\_insM()} \rightarrow \mathrm{RES}\ \} \\ \quad \mathrm{abs(RES)} = \mathrm{sort}(\overline{\mathrm{S}}) \wedge \mathrm{RES} = \mathrm{SLF} \end{array}$$

The proofs of *sortISDIRP_INV* and *sortTOP_INV* are straightforward using analog lemmas for sorted_ins. The proof of *sorted_insLEMMA* takes more care, but uses the same proof technique as shown above.

[3] I.e. the Hoare-like logic for the object-oriented programming language.

[4] The equalities hold for all $\overline{\mathrm{S}}$ with $\neg$ isempty[$\overline{\mathrm{S}}$] which is a trivial invariant to the proof.

4 Implementations

In the preceding sections, we looked at software moduls with the eyes of a client, i.e. the module/class specification was used to understand the meaning of the procedures/methods and to prove client programs correct. This section discusses the implementation side. First, it demonstrates that our interface specification method is sufficiently abstract and flexible to capture different implementations; as example we will sketch two list implementations satisfying the given interface specification for CList. Along with the first example, we will demonstrate how the abstraction and sharing functions are specified. Finally, we will shortly discuss the problem of proving implementations correct.

Implementation by Singly Linked Lists As first implementation example, we use the common implementation of lists by singly linked objects with two instance variables. Here we only provide the implementation of restM, updtopM, and concM (for a graphical illustration of concM confer section 2.1). The implementation of the other methods are assumed to be understood.

```
class implementation  CList

  var head: Elem
  var tail: CList

  // implementation of methods
  //  emptyM, isemptyM, topM(), appM, copyM

  method restM(): CList
     SELF.tail
  end

  method updtopM( E: Elem ): CList
     SELF.head := E ;
     SELF
  end

  method concM( L: CList ):  CList
     if  SELF.isemptyM()  then  L
     else
        CList LVAR := SELF ;
        while  LVAR.restM().isemptyM().not  do
           LVAR :=  LVAR.restM()
        end ;
        LVAR.tail :=  L ;
        SELF
     end
  end
```

Along with a class implementation, the implementor should provide the specifications of the abstraction and sharing functions. These specifications are needed to prove that an implementation satisfies an interface specification. Abstraction and sharing functions are specified in the interface specification language. For this and similar tasks, interface specification languages usually provide the syntax of side-effect free expressions of the corresponding programming language (cf. [GH91]). In our example language, side-effect free expressions are either constants, or identifiers possibly followed by a sequence of attribute selections; e.g. L.tail.head. There are several aspects where interface specification languages need to refer to variables, i.e. to the often so-called L-value of an expression (e.g. in many interface specification languages a procedure specification contains a modifies clause specifying which variables are allowed to be modified by the procedure). The L-value of an expression E is denoted by var(E); i.e. evaluating E yields the contents of variable var(E). With this notation the specifications of abs, isdirpart, and isdisj are straightforward:

abs : CList → List
abs(L) =
 if L = null then empty
 else app(L.head, abs(L.tail))

isdirpart : CList x CList → Bool
isdirpart(L1, L2) = (L1 = L2.tail)

isdisj : CList x CList → Bool
isdisj(L1, L2) =
 (varsof(L1) ∩ varsof(L2) = emptyset)

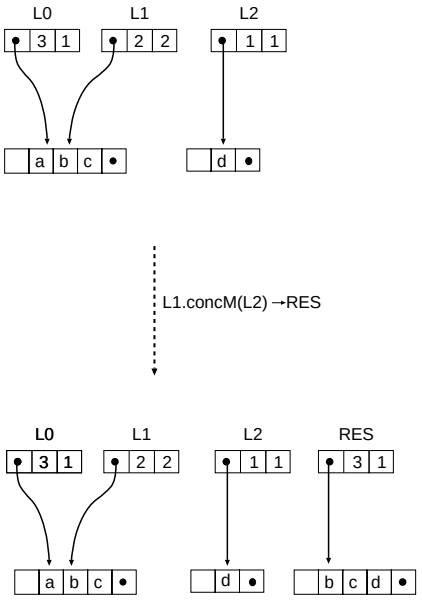
varsof: CList → Set(Variable)
varsof(L) =
 if L = null then emptyset
 else { var(L.head) } ∪ varsof(L.tail)

The above specifications make obvious that abstraction and sharing functions depend on the class/module implementation (e.g. not all implementations provide the attributes head and tail). Connecting implementations with interface specifications, abstraction and sharing functions are indispensable for proving implementations correct. In particular, they can be used to prove the program independent properties of the interface specification like *isdirpartABS* and *isdisjCOMM* (cf. end of section 2.1).

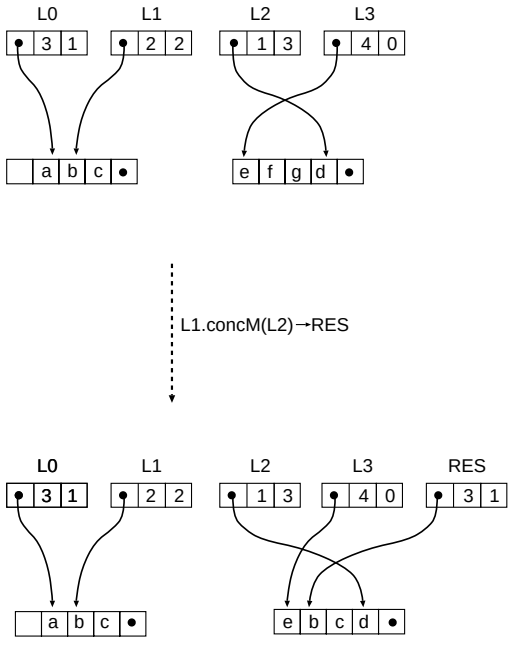
Array Implementation To show that very different implementations can satisfy a common interface specification, we sketch here another list implementation based on arrays that also satisfies the axioms given in section 2.1. A list is represented by a triple (PTR,LENGTH,AVAILABLE_CELLS) where PTR is a pointer into a dynamic array[0..N] of type Elem, LENGTH is the length of the list and AVAILABLE_CELLS is the index of the element that PTR points to[5]. How the methods emptyM, isemptyM,

[5] Instead of this C like implementation that better fits our needs for graphical illustration, we could have used a triple (ARRAY,FIRSTINDEX,SIZE).

topM, copyM, and updtopM can be implemented for such representations should need no explanation. The implementation for concM is illustrated by the following figures. There are two cases: If the length of the first list L1 is greater than the number of available cells of list L2, a big enough, new array and a new triple object are allocated; the relevant parts of the old arrays are copied into the new one, and the new triple object is initialized accordingly:



If the length of the first list L1 is less than or equal to the number of available cells of list L2, the relevant part of the array of L1 is copied into the available cells of L2, possibly overwriting lists shared with L2; for the result list a new triple is returned:



To implement restM according to our interface specification, restM has to allocate a new triple object and has to initialize PTR to the incremented pointer of its SELF parameter. According to the invariant axiom for appM (see appendix), we have to copy the SELF parameter in the implementation of appM, because the implementation cannot know whether the available cells are in use. Of course, we can refine this implementation by adding such information or by adding a second append method that works by selective update similar to concM.

There are quite a number of other sensible list implementations satisfying our interface specification (e.g. doubly linked lists). For each implementation, we can produce a fine grain interface specification that exactly reflects the sharing properties of that particular implementation. As these interface specifications are sets of theorems in our framework, we can order them by implication getting a hierarchy of interface specifications which can be considered as a refinement of a subtyping/inheritance hierarchy.

Discussing Correctness of Implementations Proving that an implementation satisfies an interface specification is in general more difficult than proving the correctness of client programs: Whereas in client programs all updating is captured by the interface specification in a rather abstract way, implementations have to establish these abstractions and they have to use updates of object attributes, like e.g. "LVAR.tail := L" in the body of concM above. Whereas assignments to local identifiers, like the assignment "LVAR := LVAR.restM()" in the body of concM, have no effect on the object structures and can be logically handled by Hoare's assignment law, assignments to object attributes usually produce side-effects, and it is not trivial to prove that a proposition is invariant under the assignment: E.g. the proposition abs(SLF) = $\overline{S}$ is in general not invariant under "LVAR.tail := L" (and in particular not in the context of concM), although the identifier SLF does not occur in the assignment. The reason for this is that the value of expression "abs(SLF)" depends not only on the object that is denoted by SLF, but as well on all instance variables reachable from SLF.

To provide proof rules as well for this kind of assignments, we are currently working on adapting the approach to pointer algebras developed by B. Möller [Mö93] to our framework. This means that we have to make parts of the state accessible in the proof rules. The prerequisites for this are taken from the specification of the underlying programming language. We use operational language specifications based on an algebraic framework. This framework allows in particular to explicitly specify the set of variables (cf. [PH94b]). In addition to this, the higher-level proof rules, we are looking for, can be proved correct w.r.t. the operational programming language semantics (cf. [PH94a]).

5 Conclusions

We have presented a new method for specifying interfaces of program modules. Compared to existing approaches, the method has two advantages:

1. It supports the abstract specification of sharing and selective updates in object structures.
2. It can be directly applied to prove correctness of client programs.

We believe that supporting sharing and selective updates is an important step towards making specification and verification techniques for programs more practical. Even in applications in which we are in the end only interested in modules with side-effect free procedures, the presented specification and verification technique can help to prove such implementations correct.

References

[Bre91] R. Breu. *Algebraic Specification Techniques in Object–Oriented Programming Environments*, volume 562 of *LNCS*. Springer–Verlag, 1991.

[CL94] Y. Cheon and G. T. Leavens. A quick overview of Larch/C++. *Journal of Object–Oriented Programming*, 7(6):39–49, October 1994.

[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*, 1991. LNCS 552.

[GH93] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[Jon89] H. B. M. Jonkers. An introduction to COLD–K. In M. Wirsing and J.A. Bergstra, editors, *Algebraic methods: theory, tools and applications*, volume 394 of *LNCS*, pages 139–206. Springer–Verlag, 1989.

[Jon91] H. B. M. Jonkers. Upgrading the pre– and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 428–456. Springer–Verlag, 1991.

[Mö93] B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21:57–90, 1993.

[PH94a] A. Poetzsch-Heffter. Deriving partial correctness logics from evolving algebras. In B. Pehrson and I. Simon, editors, *Proceedings of the IFIP 13th World Computer Congress '94*. Elsevier, August 1994.

[PH94b] A. Poetzsch-Heffter. Developing efficient interpreters based on formal language specifications. In P. Fritzson, editor, *Compiler Construction*, 1994. LNCS 786.

Appendix

Interface specification for appM and updtopM:

SLF.appM(E) → RES
pre abs(SLF) = $\overline{\text{SLF}}$
ens abs(RES) = app(E,$\overline{\text{SLF}}$)

SLF.appM(E) → RES
inv true

SLF.appM(E) → RES
pre isdisj[L,SLF]
ens isdisj[L,RES]

SLF.updtopM(E) → RES
pre ¬ isempty[abs(SLF)] ∧ abs(SLF) = $\overline{\text{SLF}}$
∧ abs(L) = $\overline{\text{L}}$ ∧ (ispart[SLF,L] ∨ SLF = L)
∧ $\overline{\text{L}}$ = conc($\overline{\text{PREFIX}}$,$\overline{\text{SLF}}$)
ens abs(L) = conc($\overline{\text{PREFIX}}$,app(E,rest($\overline{\text{SLF}}$))

SLF.updtopM() → RES
pre ¬ isempty[abs(SLF)]
inv L: isdisj[L,SLF] ∨ ispart[L,SLF]

SLF.updtopM() → RES
pre ¬ isempty[abs(SLF)]
ens RES = SLF

SLF.updtopM(E) → RES
pre ¬ isempty[abs(SLF)] ∧ isdirpart[L1,L2]
ens isdirpart[L1,L2]

The above specification axioms use the sharing predicate ispart, which is the transitive closure of isdirpart.

Finally, we give the signature of the abstract data type LIST refered to in the paper:

spec LIST is
uses Elem
sorts List
empty: → List
isempty: List → Bool
top: List → Elem
rest: List → List
app: Elem x List → List
conc: List x List → List
sort: List → List
end spec