


**REASONING ABOUT
DYNAMICALLY EVOLVING PROCESS
STRUCTURES**

**A PROOF THEORY FOR
THE PARALLEL OBJECT-ORIENTED LANGUAGE
POOL**

Frank Sipke de Boer

 **Bibliotheek**
Centrum voor Wetenschap en Informatie
informatie

VRIJE UNIVERSITEIT TE AMSTERDAM

REASONING ABOUT DYNAMICALLY EVOLVING
PROCESS STRUCTURES

A PROOF THEORY FOR THE PARALLEL OBJECT-ORIENTED LANGUAGE
POOL

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. C. Datema,
hoogleraar aan de faculteit der letteren,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der wiskunde en informatica
op maandag 15 april 1991 te 15.30 uur
in het hoofdgebouw van de universiteit, De Boelelaan 1105

door
Frank Sipke de Boer
geboren te Groningen

Centrum voor Wiskunde en Informatica
1991

Promotor : prof. dr. J.W. de Bakker
Copromotor : dr. P.H.M. America
Referent : prof. dr. W. Damm

Acknowledgements

First of all, I would like to thank Jaco de Bakker for providing me the opportunity of doing research at such a scientifically stimulating and socially pleasant environment as the CWI. Furthermore, I want to acknowledge Jaco's persistent insistence on a high quality of the presentation without which most of this work would be a rather amorphous mass of technicalities.

But for this latter point I am equally obliged to my copromotor Pierre America. I thank Pierre for the unbending patience he showed in carefully reading this thesis and his innumerable suggestions all of which contributed to a considerable improvement of the original manuscript.

Werner Damm I would like to thank for refereeing this thesis, and for his interest and appreciation. I do not have the illusion I will ever have the occasion again of presenting my work for about six hours in succession for such a keen and alert listener.

I am grateful to Willem-Paul de Roever for his interest and encouragements.

I thank the members of the Amsterdam Concurrency Group: Jaco de Bakker, Franck van Breugel, Arie de Bruin, Jean-Marie Jacquet, Peter Knijnenburg, Joost Kok, Jan Rutten, Erik de Vink and Jeroen Warmerdam, for providing the opportunity to discuss one's preliminary ideas.

Much of the work presented in this thesis I was also able to discuss in the Working Group on Semantics of ESPRIT project 415, for which I am grateful.

With respect to my general scientific development I greatly profited from my collaboration with Catuscia Palamidessi.

With Joost Kok, Catuscia Palamidessi and Jan Rutten I am very fortunate to enjoy such a fruitful merging together of scientific research and intimate friendship.

I want to thank Juditha Melssen, it is due to her that I found the motivation to finish my studies and start a scientific career. Also I thank our daughter Yentl for being such a fine little understanding person.

Finally, I am grateful to my mother Dina de Boer-Pilon for her support and to my brother Erik de Boer, and Jan and Lies Melssen, for their encouragements.

Contents

1 Introduction	1
2 A proof theory for a sequential version of POOL	15
2 A proof theory for process creation	121
3 A proof theory for the language POOL	201
Samenvatting	269

1 Introduction

1.1 The correctness problem

An important field of research in computer science is the development of deductive systems for proving formally that a program is correct.

With respect to the correctness problem the class of programming languages of an *imperative* nature has enjoyed special attention in the literature. These languages provide an abstraction of the Von Neumann model of a computer. A computer according to this model consists of essentially two units: a *memory* and a *processor*. The processor retrieves some information from the memory, performs some operations upon this information and stores the result back into the memory. From an abstract point of view the basic operations of a processor thus consist of reading from the memory and writing to it. This view is reflected by the basic instruction of an imperative language, i.e., the *assignment*. Essentially such an assignment has the form $x := e$, where x is a *variable*, and e an *expression*. The notion of a variable provides an abstraction from the memory. The execution of an assignment instruction can be described as follows: First, the expression e is “evaluated”, this evaluation can be viewed as an abstraction from the sequence of actions consisting of

1. retrieving the required information from the memory, and
2. processing this new information.

Next the value of e is assigned to the variable x , which corresponds to storing the result into the memory. Programs are constructed from assignments using operators which describe the “flow of control”.

A consequence of the Von Neumann model is that with respect to imperative programming languages we have a distinction between a program and its specification, which tells *what* the program is supposed to achieve. Formally, such a specification is a mathematical statement of a logic, for example, a first-order logic. This distinction then gives rise to the problem of how to prove formally that a program meets (*implements*) a particular specification, or in other words, that the program is *correct*.

1.2 Hoare-style proof-systems

There have been developed several ways to reason about the correctness of a program. The formalization of the reasoning about the correctness of a program first depends

upon the class of properties of a program one is interested in. The formal specification of such a class consists in defining an appropriate *semantics* for the programming language. Such a semantics describes the behaviour of a program which is of relevance for determining whether a program satisfies a certain property.

An important notion of correctness is what is called the *partial correctness* ([Ho1]) of a program. This notion of correctness is based upon the interpretation of a program as a *state transformation*, a state being an assignment of *values* to the *variables* of the programming language. This concept of a state provides an abstraction of the memory of the underlying machine. The state transformation associated with a program then tells us, given an initial state, what the result will be of the execution of the program. The partial correctness of a program specifies that whenever the execution of a program starts in a state satisfying some property then upon termination a certain property will hold. These properties of states are formulated usually in some kind of first-order logic, they express some relation between the values of the variables of a state.

Given some *assertion* language to describe a state, the partial correctness of a program S is specified formally by a Hoare triple, i.e., a construct of the form

$$\{p\}S\{q\}$$

where p and q are formulas of the assertion language. Such a specification is interpreted as

every terminating execution of the program S starting from a state satisfying p will result in a state satisfying q .

The assertion p is called the *precondition*, it describes the initial state, whilst the assertion q is called the *postcondition*, it describes the final state.

To prove that a program S is correct then amounts to proving that a correctness formula $\{p\}S\{q\}$ is valid. Thus we are interested in a formal proof system for deriving valid correctness formulas. Such a proof system will consist of some *axioms*, some valid correctness formulas, and some *rules*, which enable one to derive new valid correctness formulas from already established ones. More specifically, a rule will be of the form

$$\frac{\{p_1\}S_1\{q_1\}, \dots, \{p_n\}S_n\{q_n\}}{\{p\}S\{q\}}$$

Here the correctness formulas above the line are called the *premises* whilst the correctness formula under the line is called the *conclusion* of the rule. In general the program S will be constructed from S_1, \dots, S_n and the way it is constructed from

these programs will be reflected in the way the assertions p and q are constructed in the logic from $p_1, \dots, p_n, q_1, \dots, q_n$, respectively.

In general however we have to include also a rule to incorporate in the system inferences about properties of states. Such a rule is called the *consequence* rule:

$$\frac{p \Rightarrow p_1, \{p_1\}S\{q_1\}, q_1 \Rightarrow q}{\{p\}S\{q\}}$$

Here \Rightarrow should be interpreted as implication and the assertions of the rule as valid assertions of the logic. So essentially we assume all the valid assertions given as additional axioms of the system.

This formalization of the notion of correctness has been applied to a variety of programming constructs (see, for example, [Ba]).

1.3 The proof-theory of a parallel object-oriented language

In this thesis we investigate the Hoare-style proof theory of an abstract version of a parallel object-oriented language called POOL ([Am2]). The language POOL has been designed by P. America at the Philips Research Laboratories in the context of the Esprit project 415. The semantic foundations of the language POOL have been studied in [AR]. In general, a parallel language like POOL provides an abstraction of a parallel computer, i.e., a computer consisting of a *network* of processors, each with its own local memory. The way in which the parallelism of the underlying computer is represented in the language is described below.

The result of this investigation we present in the form of three papers:

1. A sound and complete proof system for a sequential version of POOL, P. America and F. de Boer.
2. A proof system for process-creation, P. America and F. de Boer. A version of this paper appeared in the deliverable of the Esprit project 415 of the Working Group on Semantics and Proof theory, D4, October 1988, and an extended abstract of it appeared in TC2 Working Conference on Programming Concepts and Methods, Sea Gallilee, Israel, 1990.
3. A proof system for the language POOL, F. de Boer. A version of this paper will appear in Foundations of Object-Oriented Languages, J.W. de Bakker, W.P. de Roever, and G. Rozenberg (editors), Lecture Notes in Computer Science, and an extended abstract of it appeared in the proceedings of the seventeenth International Colloquium on Automata, Languages and Programming (ICALP), volume 443 of Lecture Notes in Computer Science, Warwick, England, 1990.

In the first paper of this thesis we develop a proof theory for a sequential version of the language POOL, the language SPOOL, generalizing the proof techniques for sequential programming languages with recursive procedures. We use the language SPOOL to focus on the object-oriented features of the language POOL.

In [AFR] a proof theory based on the formalism of Hoare triples has been developed for a parallel language modeled after CSP ([Ho2]). In the second paper we generalize this method to a parallel version of the language POOL, which we simply call P . To focus on the parallelism present in POOL we introduced in the language P a CSP-like communication mechanism, instead of the more complicated rendezvous mechanism of POOL.

Finally, in the last paper, we formalize reasoning about the correctness of the rendezvous mechanism of POOL. This formalization generalizes the axiomatization of the Ada rendezvous [GR]. Combining this with the result obtained in the previous paper we obtain a proof theory for an abstract version of POOL containing the main programming constructs.

The systems described by the languages studied in this thesis consist of a collection of *objects*. An object contains some private data stored in variables which are not directly accessible to other objects. Objects can interact only by means of some synchronous message passing mechanism. An interaction can be established between objects which “know” each other. This means that part of the data stored in the private memory of an object will contain *references* to other objects. These references themselves can be communicated from one object to another. Objects are grouped into classes, the behaviour of objects belonging to one class is described uniformly. An object can *create* another object belonging to a particular class. A reference to this newly created object is then added to the data of the “creator”. At the moment of its creation this new object is only known by its creator, however the creator can communicate a reference to this new object to other objects. As a consequence the programming languages studied give rise to *dynamically evolving systems of objects*.

The parallelism of the languages of the chapters two and three is introduced by allowing objects to have an activity of their own. When an object is created it starts executing in parallel with the already existing objects. On the other hand, the sequential version of POOL describes systems in which at every moment there is only one object executing. This object can pass control to another object by sending it a message. When the result of this message is returned control is passed back again to the caller.

One of the main proof-theoretical issues is how to describe such a dynamically evolving system of objects. We want to describe such systems at an abstraction level at least as high as that of the programming language itself. That is we allow the assertion language to speak about such a system in terms of only two operations on “pointers”:

- pointers can be checked for equality, and
- they can be “dereferenced”.

Given a pointer to an object the dereferencing mechanism allows to refer to the local memory of that object.

Another important feature of the assertion language is that it only allows to speak about *existing* objects, objects which do not (yet) exist can not be referred to. As a consequence the quantification ranging over objects belonging to a particular class, introduced to be able to express interesting properties of the topology, is interpreted as ranging over existing objects.

The abstraction level of the assertion language has some important consequences for the proof-theory to be developed. First, it makes necessary to axiomatize the phenomenon of *aliasing*, which is due to the existence of different expressions referring to the same variable. In the absence of aliasing the assignment instruction is axiomatized by

$$\{p[e/x]\}x := e\{p\}$$

where $p[e/x]$ denotes the assertion obtained from p by substituting every occurrence of x by the expression e . The validity of this axiom follows directly from the observation that if a state satisfies the assertion $p[e/x]$ then p will hold in the state resulting from assigning the value of e to the variable x . The substitution of the expression e for the variable x in p actually computes as what is known as the *weakest precondition* ([Dij]) of the assignment $x := e$ with respect to the postcondition p : $p[e/x]$ is implied by every assertion q such that $\{q\}x := e\{p\}$ holds. This literal replacement of x by e , however, does not work anymore if there occur in p expressions which are aliases of the variable x . The phenomenon of aliasing to which the reasoning about dynamically evolving systems gives rise can be considered as a particular case of the problem of axiomatizing reasoning about datastructures like *arrays* and (pointers to) *records*. For example, let me be a (pointer) variable referring to a record of type *person* containing a field *age*. Consider the assignment of the integer 35 to the field *age* of the record referred to by me . We want to compute the weakest precondition of this assignment with respect to the postcondition $other.age = 34$, where $other$ is another variable referring to a record of type *person*. A simple literal replacement of the expression $me.age$ by 35 in the assertion $other.age = 34$ obviously does not work because it does not take into account that the variables me and $other$ might in fact refer to the same record, in which case the weakest precondition should be *false*! Thus the computation of the weakest precondition requires an additional test for equality: if $me = other$ then we perform the substitution of the integer expression 35 for the expression $other.age$ in the assertion $other.age = 34$, which yields indeed an assertion equivalent to *false*. On the other hand if $me \neq other$ then the expression $other.age$ refers to a different variable than $me.age$, in which case the weakest precondition is

the assertion $other.age = 34$ itself. Summarizing, as weakest precondition we obtain the assertion: *if* $other = me$ *then* $35 = 34$ *else* $other.age = 34$, which is equivalent to $other \neq me \wedge other.age = 34$. A similar analysis applies to the reasoning about dynamically evolving systems of objects as described by the language POOL and about other datastructures like arrays. In this thesis a substitution mechanism is defined which generalizes the above analysis.

Another problem is the axiomatization of the creation of new objects. The problem is that a straightforward axiomatization along the lines of the assignment axiom does not work because in the state just before the creation of an object this object does not exist, so we can not refer to it! However, as the only operations on pointer variables are testing on equality and dereferencing, it is possible to define a substitution mechanism which does compute the weakest precondition without using a reference to the newly created object. Consider for example the case that a reference to the newly created object is stored in the variable x of the creator. In the assertion describing the state just after the creation x can occur essentially only in subformulas of the form $x = y$, or in subexpressions of the form $x.y$, where the variable x is dereferenced. Now, in the case that x is tested for equality, we can statically determine the outcome of this test in the state just before the creation: $x = y$ will evaluate in the new state to *false* whenever the variable y is syntactically distinct from x because the value of y is not affected by the creation of a new object and thus refers to an "old", i.e., already existing, object. Also the value of an expression like $x.y$ can be predicted in advance because the variables of a newly created object are initialized to the value *nil*, which stands for undefined. Additionally the above analysis has to take into account that the creation of an object changes the scope of the quantifiers.

Besides these problems of aliasing and object-creation, we have to axiomatize the interaction between objects. Objects in SPOOL interact by means of a *remote procedure call*. Each object has its own set of procedures (also called *methods*), which can be called upon by other objects. When an object sends a request to another object to execute a specified procedure it additionally provides the receiver of the call with the actual parameters. During the execution of the requested procedure the caller is suspended. The caller resumes its own activity upon reception of the result. The axiomatization of such a call is based upon the axiomatization of recursive procedures of sequential programming languages, the basic pattern of which can be described as follows: let P be a *procedure* variable declared as the statement S . To derive a correctness formula $\{p\}P\{q\}$ about a call of P we have to derive the corresponding correctness formula about S . But the simple rule

$$\frac{\{p\}S\{q\}}{\{p\}P\{q\}}$$

induces a circularity in case of occurrences of P in S . This circularity is overcome by requiring the derivation of $\{p\}S\{q\}$ *assuming*, i.e., introducing as an additional

axiom, $\{p\}P\{q\}$. Formally this is rendered by the following rule

$$\frac{\{p\}P\{q\} \vdash \{p\}S\{q\}}{\{p\}P\{q\}}$$

where the premise $\{p\}P\{q\} \vdash \{p\}S\{q\}$ should be interpreted as “ $\{p\}S\{q\}$ is derivable assuming as additional axiom $\{p\}P\{q\}$ ”. When the call to P is a remote procedure call the above rule has additionally to take into account the phenomenon of the “context-switch”, that is, in axiomatizing a call we have to switch explicitly from the point of view of the caller to that of the callee.

The reasoning about the interaction between objects described by the parallel languages of the chapters two and three is based on the following proof method:

The local stage At this stage objects are specified in isolation, the interaction of an object with its environment is described by means of *assumptions*, i.e., correctness formulas describing actions which depend upon the environment. At this stage we use an assertion language which speaks only about the local state of an object.

The intermediate stage This stage is usually called the *cooperation test*. It consists of checking the mutual consistency of the assumptions introduced in the previous stage. At this stage we have to reason about a system of objects; the assertion language designed for this purpose, we call the *global* assertion language.

The global stage At this stage we describe how the local specifications of the objects can be combined into a specification of the main program. Also here the global assertion language is used.

1.4 On the formal justification of the proof-theory of POOL

There are two important requirements one should like a proof system for correctness of programs to satisfy. First, we want the system to be sound, that is, all the derivable correctness formulas must be valid. To prove formally the soundness of a proof system for program correctness we have to define formally the validity of a correctness formula. To this end we have to define formally the semantics of a program. For the parallel languages studied in this thesis we define the semantics of a program by means of a transition system.

A transition system specifies an abstract machine for the execution of programs. A configuration of such a machine consists of a program and a state. Its behaviour is

specified by a relation between configurations. Given a configuration, this relation tells us what a possible next configuration is by selecting the first instruction and executing it. The resulting configuration then consists of the part of the program still to be executed and the updated state.

By means of a transition system we then define the state transformation associated with a program.

For the semantic description of the sequential language SPOOL studied in the first chapter a different approach is taken. Here the semantics of a program is defined in a compositional way, that is, the semantics of a program is defined in terms of its syntactical constituents. More precisely, the meaning of a program will be an element of a mathematical domain, in our case a complete partial order. A semantical description of the language then consists of associating with the programming constructs operators on this domain, such that the meaning of a compound program can be obtained by applying the appropriate operator to the meanings of its syntactical constituents. The mathematical structure of this domain will be used to describe the semantics of programming constructs which give rise to infinite behaviour.

Given the semantics of the programming language we define the semantics of correctness formulas in a formal way. The soundness of the proof system under consideration then is established essentially by an inductive argument on the length of the derivation. First the axioms are shown to be valid, then we show that the rules preserve validity, by proving the validity of the conclusion assuming the validity of the premises.

Another important requirement is *completeness*: Every valid correctness formula is derivable. In this thesis we use two techniques for proving completeness.

To the proof system for the language SPOOL we apply the technique developed for sequential imperative languages with recursive procedures. This technique is based upon the expressibility of the *strongest postcondition*. We assume that there exists for every statement S of the programming language and assertion p an assertion which characterizes exactly the set of final states of executions of S starting from a state satisfying p .

Given this assumption we can define for every procedure a correctness formula which describes its associated state-transformation. Such a formula is called a most general correctness formula. An inductive argument shows then that every valid correctness formula is derivable from these most general correctness formulas. As a special case, using the proof rule for recursive procedures, we have the derivability of these correctness formulas themselves, which then concludes the completeness proof.

Now the expressibility of the strongest postcondition usually follows from the fact that we can express in the assertion language some kind of coding of finite computations.

However, due to the abstract nature of the assertion language we use, the expressibility of the strongest postcondition requires a substantial generalization of the standard techniques as described in [TZ].

For the parallel languages of the chapters two and three we generalize the technique for proving the completeness of the proof system developed for CSP ([Ap]). This technique is based upon the construction of a most general *proof-outline* for each class definition. Such a proof-outline associates with each substatement S of a particular class definition assertions $pre(S)$ and $post(S)$. The assertion $pre(S)$ describes the set of local states which characterize the internal data of an object which is about to execute S in some computation of the program. Analogously, the assertion $post(S)$ describes the set of local states which characterize the internal data of an object which has just finished executing S in some computation of the program. The proof of the existence of these assertions is also non-trivial due to the abstract nature of the assertion language.

Then it is shown that these proof-outlines can be constructed by the axioms and rules of the local proof system.

Furthermore, it is shown that these proof-outlines cooperate, in the sense that they mutually validate the assertions $post(S)$, with S a statement involving object-creation, or some communication.

The main idea of this cooperation test is the use of *history* variables, i.e., variables which record for each object the sequence of its interactions with the environment. These history variables are used to express the conditions under which different computations can be merged into one computation. These conditions are described by an assertion which is called the *global invariant*.

The use of these history variables can be explained as follows: Let S_1 and S_2 be two statements involving some communication, say, S_1 describes a send action, whilst S_2 describes its corresponding receive action. In the cooperation test we then have to show that the communication between two objects α and β , as described by S_1 and S_2 in a state in which the internal data of α is described by $pre(S_1)$ and that of β by $pre(S_2)$ will result in a state in which the internal data of α is described by $post(S_1)$ and that of β by $post(S_2)$.

Now the assertion $pre(S_1)$ tells us that there exists a final state of a partial computation in which some object, with the same internal data as that of α , is about to execute S_1 . Analogously, we have the existence of a computation in which some object, with the same internal data as β is about to execute S_2 . However we want to have a *single* computation in which *at the same time* there exists two objects, one of which is about to execute S_1 , and with the same internal data as that of α , whilst the other is about to execute S_2 , and with the same internal data as that of

β . Because then we have by definition of the assertions $post(S_1)$ and $post(S_2)$ that after the communication between these two objects we have that their internal data are described by $post(S_1)$, $post(S_2)$, respectively. From which we infer that also after the communication between α and β the internal data of α is described by $post(S_1)$ and that of β by $post(S_2)$.

The existence of such a single computation we derive from the assumption that the global invariant holds. This assertion namely characterizes those states σ for which there exists a final state σ' of a partial computation such that the local history of each existing object of σ equals that of a corresponding object of σ' . If then there exists in σ an object α the internal data of which is characterized by $pre(S_1)$, we have that the local history of α in σ , and that of its corresponding object in σ' , equals that of an object which is about to execute S_1 in some computation. As the sequence of interactions with its environment completely determines the behaviour of an object (objects have no local non-determinism) we have that the local computation of the object in σ' , which corresponds to α , of the partial computation of the main program leading to the intermediate state σ' equals the one of some object of the partial computation of the main program which exists according to $pre(S_1)$. So we have that the object in σ' corresponding to α in σ is about to execute S_1 . In the same way we infer that in σ' there exists an object corresponding to β which is about to execute S_2 .

After the cooperation test is shown to be satisfied we combine the proof-outlines for the different classes, into a most general correctness formula of the main program. This correctness formula essentially describes the state transformation associated with the program. By an application of the consequence rule it is shown that every valid correctness formula of the main program is derivable from this most general correctness formula.

1.5 Conclusion

This thesis discusses a proof theory of the parallel object-oriented language POOL. The proof-theory is based on the pre/postcondition style of specifying the correctness of a program. It formalizes reasoning about dynamically evolving pointer structures at an abstraction level at least as high as that of the programming language itself. The proof methodology developed extends and generalizes the ideas applied to the proof theory of languages like CSP and Ada. This methodology, however, does not satisfy the requirement of *compositionality*: in the proof system presented the rule which allows the derivation of a specification of a program from specifications of its syntactical constituents is in fact a meta-rule, i.e., it requires reasoning about the derivability within the system. This is due to the fact that reasoning locally about statements which depend upon the environment like input/output statements is modeled by the introduction of assumptions about the behaviour of the environment

which have to be shown to be mutually consistent at a later stage. In [ZREB] it is shown how to obtain a compositional proof system by incorporating variables which record for each process its history of interactions with the environment. Further research consists of the development of a compositional proof system for the language POOL by generalizing these ideas.

Another research topic worthwhile pursuing is extending the proof methodology to other classes of properties than partial correctness. An important property is the absence of *deadlock*. Deadlock occurs when every object of the system is suspended because it tries to communicate to an object which is in its turn involved in establishing a communication with another object. For languages like CSP the proof system for partial correctness can be generalized for proving the absence of deadlock. This generalization heavily depends on the observation that for a CSP program a deadlock configuration can be statically identified and the number of them are finite. Using the proof system for partial correctness a deadlock configuration can be characterized by an assertion, the number of deadlock configurations being finite, proving absence of deadlock then amounts to proving that no state satisfies the disjunction of these assertions. The application of this methodology to prove the absence of deadlock of programs written in POOL breaks down because there is no fixed number of deadlock configurations and these configurations themselves cannot be in general identified in a purely static manner.

The properties of partial correctness and absence of deadlock are instances of what is known as the class of *safety* properties ([Lam]). The main characteristic of safety properties is that they express that during a computation “nothing bad happens”. Properties expressing *eventualities*, i.e., that during a computation something is guaranteed to happen, like termination, are called *liveness* properties. Most of the interesting properties of programs fall into one of these two classes. Proof systems for the derivation of these classes of properties are in general based upon the formalization of reasoning about the *temporal* behaviour of programs ([Pn]). The development of a temporal logic for the language POOL constitutes another challenging research topic.

1.6 Related work

The theoretical foundations of object-oriented languages constitute a field of research which still requires extensive exploration. Concerning the proof theory of parallel object-oriented languages the only related work we are aware of is [Mel], where a language similar to the abstract version of POOL studied in this thesis is tackled by some trace-based reasoning. The problem of dynamic pointer structures, however, is not dealt with explicitly.

In [Bo] there is developed a proof theory for a parallel language with process creation which formalizes reasoning about dynamically evolving topologies in terms of some

coding mechanism of processes, thus generating a difference in the abstraction level between the programming language and the assertion language.

As indicated above there still remains a lot to be done concerning the proof-theoretical foundations of object-oriented languages. We believe that the results presented in this thesis give a first insight into the proof-theoretical intricacies of the object-oriented paradigm and should provide a solid basis for further research.

References

- [Am1] P.H.M. America, F.S. de Boer: *A proof theory for a sequential version of POOL*. ESPRIT project 415A, Technical report.
- [Am2] P.H.M. America: *issues in the design of a parallel object-oriented language*. ESPRIT project 415A, Doc. No. 452, Philips Research Laboratories, Eindhoven, the Netherlands, November 1988. *Formal Aspects of Computing* 1(4), 1989, pp. 366-411.
- [AR] P.H.M. America, J.J.M.M. Rutten: *A parallel object-oriented language: design and semantic foundations*. Ph. D. thesis, Centre of Mathematics and Computer Science (C.W.I.), 1989.
- [AFR] K.R. Apt, N. Francez, W.P. de Roever: *A proof system for communicating sequential processes*, *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3, July 1980, pp. 359-385.
- [Ap] K.R. Apt: *Formal justification of a proof system for Communicating Sequential Processes*. *Journal of the ACM*, Vol. 30, No. 1, January 1983, pp. 197-216.
- [Ba] J.W. de Bakker: *Mathematical theory of program correctness*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.
- [Bo] F.S. de Boer: *A proof rule for process creation*. M. Wirsing (ed.): *Formal Description of Programming Concepts*. Proceedings of the third IFIP WG 2.2 working conference, Gl. Avernæs, Ebberup, Denmark, August 25-28, 1986, North-Holland.
- [Dij] E.W. Dijkstra: *A discipline of programming*. Prentice Hall, Englewood Cliffs, 1976.
- [GR] R. Gerth, W.P. de Roever. *A proof system for concurrent Ada programs*. *Science of Computer programming* 4, 1984, pp. 159-204.
- [Ho1] C.A.R. Hoare: *An axiomatic basis for computer programming*. *Communications of the ACM*, Vol. 12, No. 10, 1969, pp. 567-580,583.
- [Ho2] C.A.R. Hoare: *Communicating sequential processes*. *Communications of the ACM*, Vol. 21, No. 8, 1978, pp. 666-677.
- [Lam] L. Lamport: *The Hoare logic of concurrent programs*. *Acta Informatica*, Vol. 14, 1980, pp 21-37.
- [Mel] S. Meldal: *Axiomatic semantics of access type tasks in Ada*. Report No. 100, Institute of Informatics, University of Oslo, Norway, May 1986. To appear in *Distributed Computing*.

- [Pn] A. Pnueli: *The temporal logic of programs*. 18th IEEE Symposium on the Foundations of Computer Science, 1977, pp. 46-57.
- [TZ] J.V. Tucker, J.I. Zucker: *Program correctness over abstract data types, with error-state semantics*, CWI Monographs 6, North-Holland, 1988.
- [ZREB] J. Zwiers, W.P. de Roever, P. van Emde Boas: *Compositionality and concurrent networks: soundness and completeness of a proof system*. In Proceedings of the 12th ICALP, Nafplion, Greece, July 15-19, 1985, Springer LNCS 194, pp. 509-519.

A proof theory for a sequential version of POOL

Pierre America and Frank de Boer

Contents

1	Introduction	17
2	The language SPOOL	18
2.1	An informal introduction	18
2.2	The syntax	19
3	Semantics	23
3.1	Domain definitions	23
3.2	The semantic functions	26
3.3	Remarks on the semantics	33
4	The assertion language and its semantics	35
4.1	The assertion language	36
4.2	Semantics of assertions and correctness formulae	38
5	The proof system	42
5.1	Simple assignments	42
5.2	Creating new objects	47

5.3	Sending messages	56
5.4	Other axioms and rules	64
6	Completeness	66
6.1	Introduction	66
6.2	The strongest postcondition	73
6.3	Freezing the initial state	76
6.4	Invariance	81
6.5	Most general correctness formulae	83
6.6	The context switch	93
7	Conclusions	101
	References	102
A	A generalisation of the rule (MR)	103
B	Expressibility	106
B.1	Coding mappings	106
B.2	Arithmetizing Truth	107
B.3	Expressing the coding relationship	111
B.4	Expressing the strongest postcondition	112
C	A closure property of the semantics	115

1 Introduction

This document explores the possibilities of giving a Hoare-style proof system for a language, called SPOOL, which is a sequential version of the language POOL [1]. SPOOL is an object-oriented language, just like POOL, but it is sequential, so that we do not have to deal with the specific problems connected with parallelism (it turns out that the other problems are already difficult enough).

The main aspect of SPOOL that is dealt with is the problem of how to reason about *pointer structures*. In SPOOL, objects can be created at arbitrary points in a program, references to them can be stored in variables and passed around as parameters in messages. This implies that complicated and dynamically evolving structures of references between objects can occur. We want to reason about these structures on an abstraction level that is *at least as high as that of the programming language*. In more detail, this means the following:

- The only operations on “pointers” (references to objects) are
 - testing for equality
 - dereferencing (looking at the value of an instance variable of the referenced object)
- In a given state of the system, it is only possible to mention the objects that exist in that state. Objects that do not (yet) exist never play a role.

Strictly speaking, direct dereferencing is not even allowed in the programming language, because each object only has access to its own instance variables. However, for the time being we allow it in the assertion language. Otherwise, even more advanced techniques would be necessary to reason about the correctness of a program.

The above restrictions have quite severe consequences for the proof system. The limited set of operations on pointers implies that first-order logic is too weak to express some interesting properties of pointer structures (for example, the fact that it is possible to go from w to z by following a finite number of x -links). It is surely too weak to apply the standard techniques in proofs of completeness of a proof system, where arbitrarily long computation sequences are coded into a finite set of variables.

Therefore we have to extend our assertion language to make it more expressive. We considered two approaches:

- Using recursively defined predicates, by which the above “interesting” properties of pointer structures can be expressed quite easily. This approach is worked out in [2].

- Allowing the assertion language to reason about finite sequences of objects. In this way the above properties can also be expressed (but not quite so elegantly). This approach is studied in this report.

In section 2 we shall present the syntax of this language SPOOL. Then, in section 3 we shall give a denotational semantics for it. In section 4 we introduce an assertion language, using quantification over finite sequences of objects, in which properties of states in a computation can be formulated, and we formally define its semantics. After that, in section 5, we present a Hoare-style proof system for SPOOL using this assertion language. This proof system is proved to be sound with respect to the denotational semantics. In section 6 we prove the completeness of the system. Finally, in section 7, some conclusions are drawn from the present work.

2 The language SPOOL

2.1 An informal introduction

The shortest description of the language SPOOL would be that it results from omitting the *body* of each class in POOL-T [1]. The most important consequence of this is that the parallelism, present in POOL-T, disappears. But let us try to give a short, independent description of SPOOL.

The most important concept is the concept of an *object*. This is an entity containing data and procedures (*methods*) acting on these data. The data are stored in *variables*, which come in two kinds: *instance variables*, whose lifetime is the same as that of the object they belong to, and *temporary variables*, which are local to a method and last as long as the method is active. Variables can contain references to other objects in the system (or even the object under consideration itself). The object a variable refers to (its *value*) can be changed by an *assignment*. The value of a variable can also be nil, which means that it refers to no object at all.

The variables of an object cannot be accessed directly by other objects. The only way for objects to interact is by sending *messages* to each other. If an object sends a message, it specifies the receiver, a method name, and possibly some parameter objects. Then control is transferred from the sender object to the receiver. This receiver then executes the specified method, using the parameters in the message. Note that this method can, of course, access the instance variables of the receiver. The method returns a result, an object, which is sent back to the sender. Then control is transferred back to the sender which resumes its activities, possibly using this result object.

The sender of a message is *blocked* until the result comes back, that is, it cannot answer any message while it still has an outstanding message of its own. Therefore, when an object sends a message to itself (directly or indirectly) this will lead to abnormal termination of the program. This is an important difference with some other object-oriented languages, like Smalltalk-80 [6].

Objects are grouped into *classes*. Objects in one class (the *instances* of the class) share the same methods, so in a certain sense they share the same behaviour. New instances of a given class can be created at any time. There are two standard classes, *Int* and *Bool*, of integers and booleans, respectively. They differ from the other classes in that their instances already exist at the beginning of the execution of the program and no new ones can be created. Moreover, some standard operations on these classes are defined.

A program essentially consists of a number of class definitions, together with a statement to be executed by an instance of a specific class. Usually, but not necessarily, this instance is the only non-standard object that exists at the beginning of the program: the others still have to be created.

2.2 The syntax

In order to describe the language SPOOL, which is strongly typed, we use *typed* versions of all variables, expressions, etc. These types are indicated by subscripts or superscripts in this language description. Often, when this typing information is redundant, it is omitted. Of course, for a practical version of the language, a syntactical variant, in which the type of each variable is indicated by a *declaration*, is easier to use.

Assumption 2.1

We assume the following sets to be given:

- A set C of *class names*, with typical element c (this means that metavariables like c, c', c_1, \dots range over elements of the set C). We assume that $\text{Int}, \text{Bool} \notin C$ and define the set $C^+ = C \cup \{\text{Int}, \text{Bool}\}$ with typical element d .
- For each $c \in C$ and $d \in C^+$ we assume a set $IVar_d^c$ of *instance variables* of type d in class c . By this we mean that such a variable may occur in the definition of class c and that its contents will be a reference to an object of type d . The set $IVar_d^c$ will have as a typical element x_d^c .
- For each $d \in C$ we assume a set $TVar_d$ of *temporary variables* of type d , with typical element u_d .

- We shall let the metavariable n range over elements of \mathbf{Z} , the set of whole numbers.
- For each $c \in C$ and $d_0, \dots, d_n \in C^+$ ($n \geq 0$) we assume a set $MName_{d_0, \dots, d_n}^c$ of *method names* of class c with result type d_0 and parameter types d_1, \dots, d_n . The set $MName_{d_0, \dots, d_n}^c$ will have m_{d_0, \dots, d_n}^c as a typical element.

Now we can specify the syntax of our language. We start with the expressions:

Definition 2.2

For any $c \in C$ and $d \in C^+$ we define the set Exp_d^c of *expressions* of type d in class c , with typical element e_d^c , as follows:

$$\begin{aligned}
 e_d^c ::= & \quad x_d^c \\
 & \quad | \quad u_d \\
 & \quad | \quad nil_d \\
 & \quad | \quad self \quad \quad \quad \text{if } c = d \\
 & \quad | \quad true \mid false \quad \quad \text{if } d = Bool \\
 & \quad | \quad n \quad \quad \quad \text{if } d = Int \\
 & \quad | \quad e_{1_d'}^c \doteq e_{2_d'}^c \quad \text{if } d = Bool \\
 & \quad | \quad e_{1_{Int}}^c + e_{2_{Int}}^c \quad \text{if } d = Int \\
 & \quad \vdots \\
 & \quad | \quad e_{1_{Int}}^c < e_{2_{Int}}^c \quad \text{if } d = Bool \\
 & \quad \vdots
 \end{aligned}$$

The intuitive meaning of these expressions will probably be clear from section 2.1. Note that in the programming language we put a dot over the equal sign (\doteq) to distinguish it from the equality sign we use in the meta-language.

Definition 2.3

The set $SExp_d^c$ of expressions with possible *side effect* of type d in class c , with typical element s_d^c , is defined as follows:

$$\begin{aligned}
 s_d^c ::= & \quad e_d^c \\
 & \quad | \quad new_d \quad \quad \quad \text{if } d \in C \text{ (i.e., } d \neq Int, Bool) \\
 & \quad | \quad e_{0_{c_0}}^c ! m_{d, d_1, \dots, d_n}^{c_0} (e_{1_{d_1}}^c, \dots, e_{n_{d_n}}^c) \quad (n \geq 0)
 \end{aligned}$$

The first kind of side effect expression is a normal expression, which has no actual side effect, of course. The second kind is the creation of a new object. This new object will also be the value of the side effect expression. The third kind of side effect expression specifies that a message is to be sent to the object that results from e_0 , with method name m and with arguments (the objects resulting from) e_1, \dots, e_n .

Definition 2.4

The set $Stat^c$ of *statements* in class c , with typical element S^c , is defined by:

$$\begin{aligned}
 S^c ::= & x_d^c \leftarrow s_d^c \\
 & | \quad u_d \leftarrow s_d^c \\
 & | \quad s_d^c \\
 & | \quad S_1^c ; S_2^c \\
 & | \quad \text{if } e_{\text{Bool}}^c \text{ then } S_1^c [\text{else } S_2^c] \text{ fi} \\
 & | \quad \text{while } e_{\text{Bool}}^c \text{ do } S^c \text{ od}
 \end{aligned}$$

Again, the intuitive meaning of these statements will probably be clear. Note that a side effect expression s may occur in the place of a statement. This means that s is evaluated and then its value is discarded, so that only the side effect remains. If in a conditional statement the else-part is absent, the statement is interpreted as if it contained else nil_{Int} .

Definition 2.5

The set $MethDef_{d_0, \dots, d_n}^c$ of *method definitions* of class c with result type d_0 and parameter types d_1, \dots, d_n (with typical element μ_{d_0, \dots, d_n}^c) is defined by:

$$\mu_{d_0, \dots, d_n}^c ::= (u_{1d_1}, \dots, u_{nd_n}) : S^c \uparrow e_{d_0}^c$$

Here we require that the u_{id_i} are all different and that none of them occurs at the left hand side of an assignment in S^c (and that $n \geq 0$).

When an object is sent a message, the method named in the message is invoked as follows: The variables u_1, \dots, u_n (the parameters of the method) are given the values specified in the message, all other temporary variables are initialized to nil , and then the statement S is executed. After that the expression e is evaluated and its value, the result of the method, is sent back to the sender of the message, where it will be the value of the send-expression that sent the message.

Definition 2.6

The set $ClassDef_{m_1, \dots, m_n}^c$ of *class definitions* of class c defining methods m_1, \dots, m_n , with typical element D_{m_1, \dots, m_n}^c , is defined by:

$$D_{m_1, \dots, m_n}^c ::= c : \langle m_{1\bar{d}_1}^c \Leftarrow \mu_{1\bar{d}_1}^c, \dots, m_{n\bar{d}_n}^c \Leftarrow \mu_{n\bar{d}_n}^c \rangle$$

where we require that all the method names are different (and $n \geq 0$).

Definition 2.7

The set $Unit_{m_1, \dots, m_k}^{c_1, \dots, c_n}$ of *units* with classes c_1, \dots, c_n defining methods m_1, \dots, m_k , with typical element $U_{m_1, \dots, m_k}^{c_1, \dots, c_n}$, is defined by:

$$U_{m_1, \dots, m_k}^{c_1, \dots, c_n} ::= D_{1\bar{m}_1}^{c_1}, \dots, D_{n\bar{m}_n}^{c_n}$$

where $m_1, \dots, m_k = \bar{m}_1, \dots, \bar{m}_n$. We require that all the class names are different.

Definition 2.8

Finally, the set $Prog^c$ of programs in class c , with typical element ρ^c , is defined by:

$$\rho^c ::= \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_n} | c : S^c \rangle$$

Here we require that c occurs in c_1, \dots, c_n . (The symbol ' $|$ ' is part of the syntax, not of the meta-syntax.)

The interpretation of such a program is that the statement S is executed by some object of class c (the root object) in the context of the declarations contained in the unit U . In many cases (including the following example) we shall assume that at the beginning of the execution this root object is the only existing non-standard object.

Example 2.9

The following program generates prime numbers using the sieve method of Eratosthenes. We assume the following symbols:

- The class name $Sieve \in C$ (abbreviated sometimes by c_1) with instance variables $p \in IVar_{Int}^{c_1}$ and $next \in IVar_{c_1}^{c_1}$, temporary variable $q \in TVar_{Int}$ and method name $input \in MName_{c_1, Int}^{c_1}$.
- The class name $Driver \in C$ (abbreviated by c_2) with instance variables $i, bound \in IVar_{Int}^{c_2}$ and $first \in IVar_{c_1}^{c_2}$.

Then this is the program:

```

(Sieve : (input  $\Leftarrow$  (q) : if next  $\doteq$  nil
                        then next  $\leftarrow$  new;
                        p  $\leftarrow$  q
                        else if q mod p  $\neq$  0
                        then next ! input(q)
                        fi
                        fi
       $\uparrow$  self ),

Driver : { }
|
Driver : i  $\leftarrow$  2;
        first  $\leftarrow$  new;
        while i < bound
        do first ! input(i);
          i  $\leftarrow$  i + 1
        od
}
```

Figure 1 represents the system in a certain stage of the execution of the program.

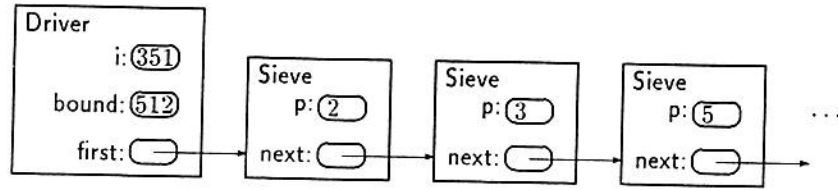


Figure 1: Objects in the sieve program in a certain stage of the execution

3 Semantics

3.1 Domain definitions

Definition 3.1

We assume for every $c \in C$ an infinite set O^c of *object names* of class c , with typical element β^c . We define P^c to be the set of all finite subsets of O^c , with typical element π^c . Furthermore we assume a function $pick^c : P^c \rightarrow O^c$ that satisfies

$$\forall \pi^c \in P^c : pick^c(\pi^c) \notin \pi^c. \quad (3.1)$$

This function will be used to generate the name of a new object, whenever one is created.

For the standard classes `Int` and `Bool` we define the sets of object names as follows:

$$\begin{aligned} O^{Int} &= Z \\ O^{Bool} &= B = \{t, f\} \end{aligned}$$

(We shall not need functions $pick^{Int}$ or $pick^{Bool}$)

Definition 3.2

For every set X we define the corresponding *flat domain* X_\perp to be the set $X \cup \{\perp\}$, equipped with an ordering \sqsubseteq defined by

$$x \sqsubseteq y \iff x = \perp \vee x = y.$$

Note that for every set X , X_\perp is a complete partial order (cpo). Sometimes we shall only consider the underlying set of this ordering, for example in definition 3.4.

Definition 3.3

We shall often use generalized Cartesian products of the form

$$\prod_{i \in A} B(i).$$

As usual, each element of this set is a function f with domain A such that $f(i) \in B(i)$ for each $i \in A$. We shall sometimes write $f_{(i)}$ for $f(i)$ if $i \in A$ and $f \in \prod_{i \in A} B(i)$, and also we sometimes write $\langle f(i) \rangle_{i \in A}$ for $\lambda(i \in A).f(i)$. Finite products are special cases: If A is of the form $\{1, \dots, n\}$ we sometimes write $B(1) \times \dots \times B(n)$.

Definition 3.4

We define the set Σ of *states*, with typical element σ , as follows:

$$\Sigma = \prod_c \mathbf{P}^c \times \prod_{c,d} (\mathbf{O}^c \rightarrow IVar_d^c \rightarrow \mathbf{O}_\perp^d) \times \prod_d (TVar_d \rightarrow \mathbf{O}_\perp^d)$$

A little explanation is surely required here. A state $\sigma \in \Sigma$ records the values of the variables in the whole system at a certain point in the computation:

- Its first component $\sigma_{(1)}$ gives for every class $c \in C$ a finite set of objects $\sigma_{(1)(c)} \in \mathbf{P}^c$. This set represents the objects that *exist* in this state (i.e., they have already been created).
- The second component $\sigma_{(2)}$ records the values of the instance variables. More concretely, if $c \in C$ and $d \in C^+$ are class names, $\beta^c \in \mathbf{O}^c$ is an object of class c , and $x_d^c \in IVar_d^c$ is an instance variable of type d in class c , then $\sigma_{(2)(c,d)}(\beta^c)(x_d^c) \in \mathbf{O}_\perp^d$ is the value of the instance variable x_d^c of object β^c . If this value is \perp , this means that the variable refers to *no* object. This is the situation for a variable that has not been initialized, but it can also be achieved by assigning nil to it.
- The third component $\sigma_{(3)}$ records the values of the temporary variables. More concretely again, if $d \in C^+$ is a class name and $u_d \in TVar_d$ is a temporary variable of type d , then $\sigma_{(3)(d)}(u_d) \in \mathbf{O}_\perp^d$ is the value of the variable u_d . Here again, it is possible that the value of the variable is nil.

For any state σ we introduce by convention that $\sigma_{(1)(\text{Int})} = \mathbf{Z}$ and $\sigma_{(1)(\text{Bool})} = \mathbf{B}$. Furthermore we write $\sigma^{(d)}$ for $\sigma_{(1)(d)}$.

Definition 3.5

Note that in general it is possible that in a state the variables of an existing object refer to an object that does *not* exist. If this is not the case and, additionally, the variables of the non-existing objects are not initialized, we say that the state is *consistent*. More precisely, we call a state σ consistent if

- $\forall c \in C \forall \beta^c \in \sigma^{(c)} \forall c' \in C \forall x_{c'}^c \in IVar_{c'}^c \quad \sigma_{(2)(c,c')}(\beta^c)(x_{c'}^c) \in \sigma_{\perp}^{(c')}$
- $\forall c \in C \forall u_c \in TVar_c \quad \sigma_{(3)(c)}(u_c) \in \sigma_{\perp}^{(c)}$
- $\forall c \in C \forall \beta^c \in \mathbf{O}^c \setminus \sigma^{(c)} \forall d \in C^+ \forall x_d^c \in IVar_d^c \quad \sigma_{(2)(c,d)}(\beta^c)(x_d^c) = \perp$

(Note that it would not make sense for either c or c' to be `Int` or `Bool`.) We shall occasionally use the shorthand $OK(\sigma)$ to indicate that σ is consistent.

Definition 3.6

We say that a state σ' *extends* a state σ (notation $\sigma \preceq \sigma'$) if $\forall c \in C \ \sigma^{(c)} \subseteq \sigma'^{(c)}$.

Definition 3.7

We shall make flexible use of the so-called *variant notation*, especially in connection with states. The variant notation is a short way to express a new state that arises when some component of an old state is modified. For example, if we write

$$\sigma' = \sigma\{\beta_1^d / \beta_2^c, x_d^c\}$$

this means the following:

$$\begin{aligned} \sigma'_{(1)} &= \sigma_{(1)} \\ \sigma'_{(2)(c,d)}(\beta_2)(x) &= \beta_1 \\ \sigma'_{(2)(c,d)}(\beta_2)(x') &= \sigma_{(2)(c,d)}(\beta_2)(x') & \text{if } x' \neq x \\ \sigma'_{(2)(c,d)}(\beta') &= \sigma_{(2)(c,d)}(\beta') & \text{if } \beta' \neq \beta_2 \\ \sigma'_{(2)(c',d')} &= \sigma_{(2)(c',d')} & \text{if } c' \neq c \text{ or } d' \neq d \\ \sigma'_{(3)} &= \sigma_{(3)} \end{aligned}$$

(This example also illustrates the usefulness of this notation.)

Definition 3.8

The set Δ^c of *contexts* of class c , with typical element δ^c , is defined as follows:

$$\Delta^c = \mathbf{O}^c \times \prod_{c'} \mathbf{P}^{c'}$$

The meaning of a context δ^c is as follows:

- The first component $\delta_{(1)}^c \in \mathbf{O}^c$ indicates the object that is currently executing (the object denoted by `self`).
- The second component $\delta_{(2)}^c$ represents all the objects that are waiting for the result of a message they have sent. This is because these objects become *blocked*, that is, they cannot answer any message before they have received the result of their outstanding message. If $c' \in C$ is a class name, then $\delta_{(2)(c')}^c \in \mathbf{P}^{c'}$ is the set of blocked objects of class c' .

Definition 3.9

We say that a context δ^c *agrees* with a state σ if

- $\delta_{(1)}^c \in \sigma^{(c)}$
- $\forall c' \in C \quad \delta_{(2)(c')}^c \subseteq \sigma^{(c')}$

We shall write the shorthand $OK(\sigma, \delta)$ to indicate that σ is consistent and δ agrees with σ .

Definition 3.10

The domain Γ of *environments*, with typical element γ , is defined as follows:

$$\Gamma = \prod_{c, d_0, \dots, d_n} \left(MName_{d_0, \dots, d_n}^c \rightarrow \left(\prod_{i=1}^n \mathbf{O}_{\perp}^{d_i} \right) \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \rightarrow \left(\Sigma_{\perp} \times \mathbf{O}_{\perp}^{d_0} \right) \right)$$

An environment γ records the meaning of the methods. More concretely, if $c \in C$ and $d_0, \dots, d_n \in C^+$ are classes, $m \in MName_{d_0, \dots, d_n}^c$ is a method name, $\bar{\beta} = \langle \beta_1^{d_1}, \dots, \beta_n^{d_n} \rangle \in \prod_{i=1}^n \mathbf{O}_{\perp}^{d_i}$ a row of objects (each possibly \perp), $\delta \in \Delta^c$ a context, $\sigma \in \Sigma_{\perp}$ a state (again possibly \perp), then $\gamma_{(c, \bar{d})}(m)(\bar{\beta})(\delta)(\sigma)$ is a pair $\langle \sigma', \beta_0 \rangle \in \Sigma_{\perp} \times \mathbf{O}_{\perp}^{d_0}$, with the intended meaning that if the method named by m is invoked with parameters $\bar{\beta}$, in the context δ (which indicates among others the object that executes the method), and starting in the state σ , then after the execution σ' will be the new state and β_0 is the result of the message. Here $\langle \sigma', \beta_0 \rangle = \langle \perp, \perp \rangle$ indicates abnormal termination or divergence.

Definition 3.11

We call an environment γ *agreement-preserving* if for every c, d_0, \dots, d_n , for every $m_{\bar{d}}^c$, for every δ^c , for every $\sigma \in \Sigma$, and for every $\bar{\beta} = \langle \beta_1^{d_1}, \dots, \beta_n^{d_n} \rangle \in \prod_{i=1}^n \sigma_{\perp}^{(d_i)}$ (note that we consider only existing objects) we have that if $OK(\sigma, \delta)$ and $\langle \sigma', \beta_{d_0} \rangle = \gamma_{(c, \bar{d})}(m)(\bar{\beta})(\delta)(\sigma)$ and $\sigma' \neq \perp$ then $OK(\sigma'), \sigma \preceq \sigma'$, and $\beta_{d_0} \in \sigma'_{\perp}^{(d_0)}$.

Note that the requirement is somewhat stronger than preservation of the agreement between state and context. We require that the new state extends the old state and that it is consistent. This automatically implies that the context δ agrees with the new state.

3.2 The semantic functions

Definition 3.12

The semantics of expressions is given by a function

$$\mathcal{E}_d^c : Exp_d^c \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \rightarrow \mathbf{O}_{\perp}^d,$$

which is defined as follows:

$$\begin{aligned}
\mathcal{E}_d^c[e_d^c](\delta)(\perp) &= \perp && \text{(from now on we assume } \sigma \neq \perp \text{)} \\
\mathcal{E}_d^c[x_d^c](\delta)(\sigma) &= \sigma_{(2)(c,d)}(\delta_{(1)})(x_d^c) \\
\mathcal{E}_d^c[u_d](\delta)(\sigma) &= \sigma_{(3)(d)}(u_d) \\
\mathcal{E}_d^c[\text{nil}_d](\delta)(\sigma) &= \perp \\
\mathcal{E}_d^c[\text{self}](\delta)(\sigma) &= \delta_{(1)} && \text{(only if } c = d \text{)} \\
\mathcal{E}_d^c[\text{true}](\delta)(\sigma) &= \mathbf{t} && \text{(only if } d = \text{Bool} \text{)} \\
\mathcal{E}_d^c[\text{false}](\delta)(\sigma) &= \mathbf{f} && \text{(only if } d = \text{Bool} \text{)} \\
\mathcal{E}_d^c[n](\delta)(\sigma) &= n && \text{(only if } d = \text{Int} \text{)} \\
\\
\mathcal{E}_d^c[e_{1d'}^c \doteq e_{2d'}^c](\delta)(\sigma) &= \mathbf{t} && \text{if } \mathcal{E}_{d'}^c[e_1](\delta)(\sigma) = \mathcal{E}_{d'}^c[e_2](\delta)(\sigma) \\
&= \mathbf{f} && \text{otherwise} \\
&&& \text{(only if } d = \text{Bool} \text{)} \\
\\
\mathcal{E}_d^c[e_{1d}^c + e_{2d}^c](\delta)(\sigma) &= \perp && \text{if } \mathcal{E}_d^c[e_1](\delta)(\sigma) = \perp \text{ or } \mathcal{E}_d^c[e_2](\delta)(\sigma) = \perp \\
&= \mathcal{E}_d^c[e_1](\delta)(\sigma) + \mathcal{E}_d^c[e_2](\delta)(\sigma) && \text{otherwise} \\
&&& \text{(only if } d = \text{Int} \text{)} \\
\\
&\vdots \\
\\
\mathcal{E}_d^c[e_{1d'}^c < e_{2d'}^c](\delta)(\sigma) &= \perp && \text{if } \mathcal{E}_{d'}^c[e_1](\delta)(\sigma) = \perp \\
&&& \text{or } \mathcal{E}_{d'}^c[e_2](\delta)(\sigma) = \perp \\
&= \mathbf{t} && \text{if } \mathcal{E}_{d'}^c[e_1](\delta)(\sigma) < \mathcal{E}_{d'}^c[e_2](\delta)(\sigma) \\
&= \mathbf{f} && \text{otherwise} \\
&&& \text{(only if } d = \text{Bool and } d' = \text{Int} \text{)} \\
\\
&\vdots
\end{aligned}$$

Although most of these equations speak for themselves, we shall give some informal explanation.

- The function $\mathcal{E}[e](\delta)$ is *strict*, that is, it will always yield \perp if it is applied to a state σ that is equal to \perp .
- The value of an instance variable is looked up in the second component of the state σ . The first component of the context δ indicates the currently active object.
- The value of a temporary variable is looked up in the third component of the state σ .

- The value of the expression `nil` is always \perp .
- The value of the expression `self` is the first component of the context δ .
- The Boolean constants `true` and `false` get the corresponding truth-values as their value.
- Integer numbers are mapped to themselves. Note that at this point we are confusing syntactic and semantic entities a little, but here this is harmless.
- The equal sign between expressions means that we test whether their values are really the same objects. Note that this is a kind of non-strict predicate, because if both sides yield \perp , the result of the equality is nevertheless `t`.
- Addition is only defined for genuine integers: If one of the two sides yields \perp the result is also \perp .
- The same is true for the relation $<$.

Definition 3.13

The semantics of expressions with possible side effect is given by the function

$$Z_d^c : SExp_d^c \rightarrow \Gamma \rightarrow \Delta^c \rightarrow \Sigma_\perp \rightarrow (\Sigma_\perp \times \mathbf{O}_\perp^d).$$

To a side effect expression $s \in SExp_d^c$, an environment $\gamma \in \Gamma$, a context $\delta \in \Delta^c$, and a state $\sigma \in \Sigma_\perp$ (the state before evaluation of the side effect expression), this function assigns a pair $\langle \sigma', \beta \rangle \in \Sigma_\perp \times \mathbf{O}_\perp^d$, consisting of the state σ' after the evaluation and the result β of this side effect expression. Here $\langle \sigma', \beta \rangle = \langle \perp, \perp \rangle$ represents abnormal termination or divergence. The function Z_d^c is defined as follows:

$$Z_d^c[s_d^c](\gamma)(\delta)(\perp) = \langle \perp, \perp \rangle \quad (\text{from now on we assume } \sigma \neq \perp)$$

$$Z_d^c[e_d^c](\gamma)(\delta)(\sigma) = \langle \sigma, \mathcal{E}_d^c[e](\delta)(\sigma) \rangle$$

$$Z_d^c[\text{new}_d](\gamma)(\delta)(\sigma) = \langle \sigma', \beta \rangle$$

$$\text{where } \beta = \text{pick}^d(\sigma^{(d)})$$

$$\sigma' = \sigma \setminus \{\sigma^{(d)}\} \cup \{\beta\} / d \setminus \{\perp / \beta, x_{d'}^d\}_{d' \in C^+, x \in IVar_d^d}$$

note that $d \in C$, i.e., $d \neq \text{Int}, \text{Bool}$

$$\begin{aligned}
\mathcal{Z}_d^c[e_{0c'}!m_{d,d_1,\dots,d_n}^c(e_{1d_1}^c,\dots,e_{nd_n}^c)](\gamma)(\delta)(\sigma) &= \langle \sigma', \beta^d \rangle \\
\text{where } \beta_0^{c'} &= \mathcal{E}_c^c[e_{0c'}](\delta)(\sigma) \\
\beta_i^{d_i} &= \mathcal{E}_{d_i}^c[e_{id_i}^c](\delta)(\sigma) \quad (i = 1, \dots, n) \\
\langle \sigma', \beta^d \rangle &= \langle \perp, \perp \rangle \quad \text{if } \beta_0 = \perp \\
&\quad (\text{from now on we assume } \beta_0 \neq \perp) \\
\delta^{c'} &= \langle \beta_0^{c'}, \delta_{(2)} \{ \delta_{(2)(c)} \cup \{ \delta_{(1)} \} / c \} \rangle \\
\langle \sigma', \beta^d \rangle &= \gamma_{(c',d,d_1,\dots,d_n)}(m)(\beta_1, \dots, \beta_n)(\delta')(\sigma)
\end{aligned}$$

Some explanation is appropriate here.

- Again, for any s , γ , and δ , $\mathcal{Z}[s](\gamma)(\delta)$ is a strict function: if the starting state σ is \perp it delivers $\langle \perp, \perp \rangle$.
- If an expression e_d^c occurs as a side effect expression, its result is computed using the function \mathcal{E}_d^c and the state is unchanged.
- The resulting object β of a new-expression is obtained by applying the function $pick^d$ to the set $\sigma^{(d)}$ of existing objects of class d . By the property listed in equation 3.1 on page 23, we know that we really get a *new* object. The new state σ' reflects the situation after the creation of this object. In its first component $\sigma'_{(1)}$ the object β is added to the set of existing objects of class d , while the other classes are unchanged (we use the variant notation to express this). The explicit initialization of the instance variables to nil would be unnecessary if we knew that σ is consistent.
- In order to evaluate a send-expression, first the destination object β_0 and the parameters β_1, \dots, β_n are computed (in the old state). Note that if the destination is \perp (i.e., nil), then the program will fail, which is represented by setting $\langle \sigma', \beta^d \rangle$ to $\langle \perp, \perp \rangle$. Otherwise a new context is created, in which the executing object is the destination of the message, and in which the sending object is added to the set of blocked objects (of the appropriate class). Then the meaning of the method m is looked up in the environment γ and, provided with the parameters, the new context and the old state, it gives us the new state and the result of the send-expression.

Definition 3.14

The semantics of statements is given by a function

$$\mathcal{S}^c : Stat^c \rightarrow \Gamma \rightarrow \Delta^c \rightarrow \Sigma_\perp \rightarrow \Sigma_\perp,$$

which is defined as follows:

$$S^c[S^c](\gamma)(\delta)(\perp) = \perp \quad (\text{from now on we assume } \sigma \neq \perp)$$

$$\begin{aligned} S^c[x_d^c \leftarrow s_d^c](\gamma)(\delta)(\sigma) &= \sigma'' \\ \text{where } \langle \sigma', \beta \rangle &= Z_d^c[s_d^c](\gamma)(\delta)(\sigma) \\ \sigma'' &= \sigma' \{ \beta / \delta_{(1)}, x \} \end{aligned}$$

$$\begin{aligned} S^c[u_d \leftarrow s_d^c](\gamma)(\delta)(\sigma) &= \sigma'' \\ \text{where } \langle \sigma', \beta \rangle &= Z_d^c[s_d^c](\gamma)(\delta)(\sigma) \\ \sigma'' &= \sigma' \{ \beta / u \} \end{aligned}$$

$$S^c[s_d^c](\gamma)(\delta)(\sigma) = (Z_d^c[s_d^c](\gamma)(\delta)(\sigma))_{(1)}$$

$$S^c[S_1; S_2](\gamma)(\delta)(\sigma) = S^c[S_2](\gamma)(\delta)(S^c[S_1](\gamma)(\delta)(\sigma))$$

$$\begin{aligned} S^c[\text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}](\gamma)(\delta)(\sigma) &= \perp && \text{if } \beta = \perp \\ &= S^c[S_1](\gamma)(\delta)(\sigma) && \text{if } \beta = \mathbf{t} \\ &= S^c[S_2](\gamma)(\delta)(\sigma) && \text{if } \beta = \mathbf{f} \end{aligned}$$

$$\text{where } \beta = \mathcal{E}[e](\delta)(\sigma)$$

$$\begin{aligned} S^c[\text{while } e \text{ do } S \text{ od}](\gamma) &= \mu \Phi \\ \text{where } \Phi : (\Delta^c \rightarrow (\Sigma_{\perp} \rightarrow \Sigma_{\perp})) &\rightarrow (\Delta^c \rightarrow (\Sigma_{\perp} \rightarrow \Sigma_{\perp})) \text{ is defined as follows:} \\ \Phi(\varphi)(\delta)(\sigma) &= \perp && \text{if } \beta = \perp \\ &= \varphi(\delta)(S^c[S](\gamma)(\delta)(\sigma)) && \text{if } \beta = \mathbf{t} \\ &= \sigma && \text{if } \beta = \mathbf{f} \\ \text{where } \beta &= \mathcal{E}[e](\delta)(\sigma) \end{aligned}$$

Here is some informal explanation:

- For any S , γ , and δ , $S[S](\gamma)(\delta)$ is a strict function from Σ_{\perp} to Σ_{\perp} .
- If an assignment to an instance variable x is done, first the right hand side is evaluated, resulting in a new state σ' (because of possible side effects), and an object β . Now the final state σ'' is constructed from σ' by modifying its second component in such a way that the object β becomes the value of the variable x .
- For an assignment to a temporary variable, essentially the same thing is done, except that the new value is stored away in the third component of the resulting state σ'' .

- If a side effect expression occurs at the place of a statement, it is evaluated and its resulting object is ignored. Only the new state is kept (this is the first component of the result of the evaluation).
- Sequential composition of statements is modelled by letting the second statement act on the state that results from the first statement.
- For a conditional statement first the condition is evaluated. Depending on that the first or the second clause is executed (or a failure is signalled).
- A while statement is modelled by taking the least fixed point of the operator Φ . This operator takes its argument φ as an approximation of the meaning of the while statement and maps it to a better approximation, obtained by unwinding the loop one more time.

Definition 3.15

The semantics of method definitions is given by a function

$$\mathcal{M}_{d_0, \dots, d_n}^c : \text{MethDef}_{d_0, \dots, d_n}^c \rightarrow \Gamma \rightarrow \left(\prod_{i=1}^n \mathbf{O}_{\perp}^{d_i} \right) \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \rightarrow (\Sigma_{\perp} \times \mathbf{O}_{\perp}^{d_0})$$

which is defined by:

$$\begin{aligned} \mathcal{M}_{d_0, \dots, d_n}^c \llbracket (u_{1d_1}, \dots, u_{nd_n}) : S^c \uparrow e_{d_0}^c \rrbracket (\gamma)(\beta_1^{d_1}, \dots, \beta_n^{d_n})(\delta^c)(\sigma) &= \langle \sigma''', \beta^{d_0} \rangle \\ \text{where } \sigma' &= \perp && \text{if } \delta_{(1)} \in \delta_{(2)(c)} \text{ or } \sigma = \perp \\ &= \langle \sigma_{(1)}, \sigma_{(2)}, \sigma'_{(3)} \rangle && \text{otherwise} \\ \sigma'_{(3)(d)}(u_d) &= \beta_i^{d_i} && \text{if } d = d_i \text{ and } u_d = u_{id_i} \\ &&& \text{for } i = 1, \dots, n \\ &= \perp && \text{otherwise} \\ \sigma'' &= S^c \llbracket S^c \rrbracket (\gamma)(\delta^c)(\sigma') \\ \beta^{d_0} &= \mathcal{E}_{d_0}^c \llbracket e_{d_0}^c \rrbracket (\delta^c)(\sigma'') \\ \sigma''' &= \perp && \text{if } \sigma'' = \perp \\ &= \langle \sigma''_{(1)}, \sigma''_{(2)}, \sigma_{(3)} \rangle && \text{if } \sigma'' \neq \perp \end{aligned}$$

Again we give an informal explanation: The first thing to be checked when a method is to be executed is whether the executing object is *blocked*, that is whether $\delta_{(1)} \in \delta_{(2)(c)}$ or whether the starting state σ is \perp . If this is the case the result of the method will be the pair $\langle \perp, \perp \rangle$ (this will come out automatically if we set σ' to \perp). Next we construct a state σ' by initializing all temporary variables to \perp , except the formal parameters, which are bound to the corresponding actual ones (that is, the variable u_{id_i} is set to $\beta_i^{d_i}$). In this modified state σ' we execute the statement S^c of the method, which

results in a new state σ'' . In this state we can evaluate the result expression $e_{d_0}^c$, which gives us the object β^{d_0} . The state σ''' after the method execution is obtained by restoring the temporary variables to their values before the method execution.

Definition 3.16

The semantics of class definitions is given by a function

$$C_{m_1, \dots, m_n}^c : \text{ClassDef}_{m_1, \dots, m_n}^c \rightarrow \Gamma \rightarrow \Gamma$$

which is defined as follows:

$$\begin{aligned} C_{m_1, \dots, m_n}^c \llbracket c : \langle m_{1\bar{d}_1}^c \Leftarrow \mu_{1\bar{d}_1}^c, \dots, m_{n\bar{d}_n}^c \Leftarrow \mu_{n\bar{d}_n}^c \rangle \rrbracket(\gamma) \\ = \gamma \left\{ \mathcal{M} \llbracket \mu_{1\bar{d}_1}^c \rrbracket(\gamma) / m_{1\bar{d}_1}^c \right\} \dots \left\{ \mathcal{M} \llbracket \mu_{n\bar{d}_n}^c \rrbracket(\gamma) / m_{n\bar{d}_n}^c \right\} \end{aligned}$$

This means that in the environment γ the value associated with each method m in the class definition is replaced by the value obtained from the corresponding method definition. However, this method definition is still evaluated with respect to the old environment γ . Note that the order of the replacements does not matter, because it is required that all method names must be different.

Definition 3.17

The semantics of units is given by a function

$$\mathcal{U}_{m_1, \dots, m_k}^{c_1, \dots, c_n} : \text{Unit}_{m_1, \dots, m_k}^{c_1, \dots, c_n} \rightarrow \Gamma \rightarrow \Gamma$$

which is defined by:

$$\begin{aligned} \mathcal{U}_{m_1, \dots, m_k}^{c_1, \dots, c_n} \llbracket D_{1\bar{m}_1}^{c_1}, \dots, D_{n\bar{m}_n}^{c_n} \rrbracket(\gamma) &= \gamma' \\ \text{where } \gamma' &= \gamma \{ \zeta_j / m_j \}_{j=1}^k \\ \langle \zeta_1, \dots, \zeta_k \rangle &= \mu \Psi \\ \Psi(\zeta'_1, \dots, \zeta'_k) &= \langle \gamma''(c_i, \bar{d}_j)(m_j) \rangle_{j=1}^k \\ \gamma'' &= \mathcal{C} \llbracket D_1 \rrbracket \circ \dots \circ \mathcal{C} \llbracket D_n \rrbracket (\gamma \{ \zeta'_j / m_j \}_{j=1}^k) \end{aligned}$$

(we suppose that $m_j = m_{j\bar{d}_j}^{c_{ij}}$).

The main point in this definition is the construction of an environment γ' from the least fixed point of the operator Ψ . This operator Ψ takes as its argument a row $\zeta'_1, \dots, \zeta'_k$ of possible meanings of the methods defined in the unit. Assuming these meanings for the corresponding methods, a new environment γ'' is determined from the class definitions in the unit and from this environment the new meanings for the previous methods are extracted, yielding the output of Ψ . The least fixed point of Ψ therefore consists of the meanings of the methods defined in the unit, where for the other methods the meanings recorded in γ are assumed.

Because we require that all the class names (the c_i) are different, each $\mathcal{C}[\![D_i^{c_i}]\!]$ modifies a different part of the environment $\gamma\{\zeta'_j/m_j\}_{j=1}^k$. Therefore the order in which they are composed does not matter. We cannot simply take the least fixed point of $\mathcal{C}[\![D_1]\!] \circ \dots \circ \mathcal{C}[\![D_n]\!]$ because we want to preserve the meanings of methods not defined in D_1, \dots, D_n . This is important in the soundness of the recursion rule.

Definition 3.18

Finally we give the semantics of programs by defining a function

$$\mathcal{P}^c : Prog^c \rightarrow \Gamma \rightarrow \Delta^c \rightarrow \Sigma_\perp \rightarrow \Sigma_\perp$$

as follows:

$$\begin{aligned} \mathcal{P}[\![U_{m_1, \dots, m_k}^{c_1, \dots, c_n} \mid c : S^c]\!](\gamma) &= S^c[\![S]\!](\gamma') \\ \text{where } \gamma' &= \mathcal{U}[\![U]\!](\gamma) \end{aligned}$$

If every method used in the program is defined in the unit then the meaning is independent of the environment γ . One could take the “empty” environment γ_0 , defined by

$$\gamma_0(c, \vec{d})(m_d^c)(\vec{\beta})(\delta^c)(\sigma) = \langle \perp, \perp \rangle$$

(this is certainly agreement-preserving).

3.3 Remarks on the semantics

In the foregoing definition of the semantic functions that play a role in our language, we have omitted some details. One of these details is the fact that all the functions of which we need the least fixed point are indeed continuous.

Lemma 3.19

The function Φ , used in the semantics of while statements in definition 3.14, is continuous.

Proof

First of all it is easy to see that

- For every expression e_d^c and for every context δ^c , the function $\mathcal{E}[\![e]\!](\delta)$ is *strict*, i.e., that $\mathcal{E}[\![e]\!](\delta)(\perp) = \perp$.
- For every statement S^c , for every environment γ , and for every context δ^c , the function $S[\![S]\!](\gamma)(\delta)$ is also strict, i.e., $S[\![S]\!](\gamma)(\delta)(\perp) = \perp$.

Now after a little calculation it becomes clear that this is all we need to ensure the continuity of Φ , which moreover maps strict functions in $\Delta \rightarrow (\Sigma_{\perp} \rightarrow \Sigma_{\perp})$ again to strict functions (so its least fixed point will also be a strict function). \square

Lemma 3.20

The function Ψ , used in definition 3.17 to define the semantics of units, is continuous.

Proof

The proof of this lemma is somewhat more involved. It would proceed in the following steps:

- For any side effect expression s_d^c , $Z[s]$ is a continuous function from Γ to $\Delta^c \rightarrow \Sigma_{\perp} \rightarrow (\Sigma_{\perp} \times \mathbf{O}_{\perp}^d)$.
- For any statement S^c , $S[S]$ is a continuous function from Γ to $\Delta^c \rightarrow \Sigma_{\perp} \rightarrow \Sigma_{\perp}$.
- For any method definition μ_{d_0, \dots, d_n}^c , $\mathcal{M}[\mu]$ is a continuous function from Γ to $(\prod_{i=1}^n \mathbf{O}_{\perp}^{d_i}) \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \rightarrow (\Sigma_{\perp} \times \mathbf{O}_{\perp}^{d_0})$.
- For any class definition D_m^c , $\mathcal{C}[D]$ is a continuous function from Γ to Γ .
- Now we can prove that Ψ is a continuous function.

\square

In retrospect we can change the domain assignments of several entities as follows (where \xrightarrow{c} stands for continuous functions and \xrightarrow{s} for strict functions):

$$\begin{aligned}
 \Gamma &= \prod_{c, d_0, \dots, d_n} \left(MName_{d_0, \dots, d_n}^c \rightarrow \left(\prod_{i=1}^n \mathbf{O}_{\perp}^{d_i} \right) \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \xrightarrow{s} (\Sigma_{\perp} \times \mathbf{O}_{\perp}^{d_0}) \right) \\
 \mathcal{E}_d^c &: Exp_d^c \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \xrightarrow{s} \mathbf{O}_{\perp}^d \\
 \mathcal{Z}_d^c &: SExp_d^c \rightarrow \Gamma \xrightarrow{c} \Delta^c \rightarrow \Sigma_{\perp} \xrightarrow{s} (\Sigma_{\perp} \times \mathbf{O}_{\perp}^d) \\
 S^c &: Stat^c \rightarrow \Gamma \xrightarrow{c} \Delta^c \rightarrow \Sigma_{\perp} \xrightarrow{s} \Sigma_{\perp}, \\
 \mathcal{M}_{d_0, \dots, d_n}^c &: MethDef_{d_0, \dots, d_n}^c \rightarrow \Gamma \xrightarrow{c} \left(\prod_{i=1}^n \mathbf{O}_{\perp}^{d_i} \right) \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \xrightarrow{s} (\Sigma_{\perp} \times \mathbf{O}_{\perp}^{d_0}) \\
 \mathcal{C}_{m_1, \dots, m_n}^c &: ClassDef_{m_1, \dots, m_n}^c \rightarrow \Gamma \xrightarrow{c} \Gamma \\
 \mathcal{U}_{m_1, \dots, m_k}^{c_1, \dots, c_n} &: Unit_{m_1, \dots, m_k}^{c_1, \dots, c_n} \rightarrow \Gamma \xrightarrow{c} \Gamma \\
 \mathcal{P}^c &: Prog^c \rightarrow \Gamma \xrightarrow{c} \Delta^c \rightarrow \Sigma_{\perp} \xrightarrow{s} \Sigma_{\perp}
 \end{aligned}$$

Lemma 3.21

Now we come to the issues of consistent states and agreement between context and state. We can make the following observations:

- For any expression e_d^c , for any state $\sigma \in \Sigma$, and for any context $\delta \in \Delta^c$ such that $OK(\sigma, \delta)$, we have that $\mathcal{E}[e](\delta)(\sigma) \in \sigma_{\perp}^{(d)}$.
- For any side effect expression s_d^c , for any agreement-preserving environment γ , for any state $\sigma \in \Sigma$, and for any context $\delta \in \Delta^c$ such that $OK(\sigma, \delta)$, we have that if $\langle \sigma', \beta^d \rangle = \mathcal{Z}[s](\gamma)(\delta)(\sigma)$ and $\sigma' \neq \perp$ then $OK(\sigma')$, $\sigma \preceq \sigma'$ (therefore also $OK(\sigma', \delta)$), and $\beta^d \in \sigma'^{(d)}$.
- For any statement S^c , for any agreement-preserving environment γ , for any state $\sigma \in \Sigma$, and for any context $\delta \in \Delta^c$ such that $OK(\sigma, \delta)$, if $\sigma' = \mathcal{S}[S](\gamma)(\delta)(\sigma)$ and $\sigma' \neq \perp$ then $OK(\sigma')$ and $\sigma \preceq \sigma'$ (therefore also $OK(\sigma', \delta)$).
- For any method definition μ_{d_0, \dots, d_n}^c , for any agreement-preserving environment γ , for any state $\sigma \in \Sigma$, for any context $\delta \in \Delta^c$ such that $OK(\sigma, \delta)$, and for any row of (existing) objects $\bar{\beta} = \langle \beta_1^{d_1}, \dots, \beta_n^{d_n} \rangle \in \prod_{i=1}^n \sigma_{\perp}^{(d_i)}$ we have if $\langle \sigma', \beta^{d_0} \rangle = \mathcal{M}[\mu](\gamma)(\bar{\beta})(\delta)(\sigma)$ and $\sigma' \neq \perp$ then $OK(\sigma')$, $\sigma \preceq \sigma'$ (so also $OK(\sigma', \delta)$), and $\beta^{d_0} \in \sigma'^{(d_0)}$.
- For any class definition D and for any agreement-preserving environment γ we have that $\mathcal{C}[D](\gamma)$ is again an agreement-preserving environment.
- For any unit U and for any agreement-preserving environment γ we have that $\mathcal{U}[U](\gamma)$ is again an agreement-preserving environment.
- For any program ρ^c , for any agreement-preserving environment γ , for any state $\sigma \in \Sigma$, and for any context δ^c such that $OK(\sigma, \delta)$, if $\sigma' = \mathcal{P}[\rho](\gamma)(\delta)(\sigma)$ and $\sigma' \neq \perp$ then $OK(\sigma')$ and $\sigma \preceq \sigma'$ (therefore also $OK(\sigma', \delta)$).

Proof

The proof consists of an easy induction on the structure of the syntactical object under consideration. \square

4 The assertion language and its semantics

In this section we shall develop a formalism for expressing certain properties of states, and we shall give a semantics for it.

One element of this assertion language will be the introduction of *logical variables*. These variables may not occur in the program, but only in the assertion language. Therefore we are always sure that the value of a logical variable can never be changed by a statement. Apart from a certain degree of cleanliness, this has the additional

advantage that we can use logical variables to express the constancy of certain expressions (for example in the proof rule (MI) for message passing in definition 5.24). Logical variables also serve as bound variables for quantifiers.

The set of expressions in the assertion language is larger than the set of programming language expressions not only because it contains logical variables, but also because it is allowed to refer to instance variables of other objects. Furthermore we include conditional expressions in the assertion language because they are very convenient (e.g., in the axiom (SAI), see definitions 5.6 and 5.7).

In two respects our assertion language differs from the usual first-order predicate logic: Firstly, the range of quantifiers is limited to the *existing*, non-nil objects in the current state. With respect to the classes `Int` and `Bool` this only means that the range does *not* include \perp . This does not affect essentially the expressive power of the assertion language, but in most practical cases one wants to exclude \perp from the quantification, so in these cases the assertions become shorter. For the other classes this restriction means that we cannot talk about objects that have not yet been created, even if they could be created in the future. This is done in order to satisfy the requirements on the proof system stated in the introduction. Because of this the range of the quantifiers can be different for different states. More in particular, a statement can change the truth of an assertion even if none of the program variables accessed by the statement occurs in the assertion, simply by creating an object and thereby changing the range of a quantifier. (The idea of restricting the range of quantifiers was inspired by [8].)

Secondly, in order to strengthen the expressiveness of the logic, it is augmented with quantification over finite sequences of objects. It is quite clear that this is necessary, because simple first-order logic is not able to express certain interesting properties.

4.1 The assertion language

Definition 4.1

For each $d \in C^+$ we introduce the symbol d^* for the type of all finite sequences with elements from d , we let C^* stand for the set $\{d^* | d \in C^+\}$, and we use C^\dagger , with typical element a , for the union $C^+ \cup C^*$.

We define O^{d^*} to be the set of finite sequences with elements from O_\perp^d (note that the elements can also be \perp). The empty sequence ϵ^d is also included in O^{d^*} . The elements in a sequence are always numbered starting from 1. In order to simplify some formulae we define $O_\perp^{d^*}$ to be the same as O^{d^*} , in deviation from definition 3.2. In addition to β^{d^*} , we shall sometimes use α^{d^*} to range over elements of O^{d^*} .

We have the following functions:

- $len^d : \mathbf{O}^{d*} \rightarrow \mathbf{Z}$ returns the number of elements in the sequence.
- $elt^d : \mathbf{O}^{d*} \times \mathbf{Z} \rightarrow \mathbf{O}_\perp^d$ extracts from the first argument the element numbered by the second argument. If the second argument is "out of bounds" (less than 1 or greater than the length of the first argument) then the result is \perp .

Assumption 4.2

We assume that for every a in C^\dagger we have a set $LVar_a$ of logical variables of type a , with typical element z_a .

Definition 4.3

We the set $LExp_a^c$ of logical expressions of type a in class c , with typical element l_a^c , as follows:

$$\begin{array}{ll}
 l_a^c ::= e_a^c & \text{if } a \in C^+ \\
 \quad | z_a & \\
 \quad | l_{c'}^c . x_{a'}^{c'} & \text{if } a \in C^+ \\
 \quad | \text{if } l_{0_{\text{Bool}}}^c \text{ then } l_{1_a}^c \text{ else } l_{2_a}^c \text{ fi} & \text{if } a \in C^+ \\
 \quad | l_{1_d}^c \doteq l_{2_d}^c & \text{if } a = \text{Bool} \\
 \quad | l_{1_{\text{Int}}}^c + l_{2_{\text{Int}}}^c & \text{if } a = \text{Int} \\
 \quad | \vdots & \\
 \quad | l_{1_{\text{Int}}}^c < l_{2_{\text{Int}}}^c & \text{if } a = \text{Bool} \\
 \quad | \vdots & \\
 \quad | |l_d^c| & \text{if } a = \text{Int and } d \in C^+ \\
 \quad | l_{1_a}^c \cdot l_{2_{\text{Int}}}^c & \text{if } a \in C^+
 \end{array}$$

Note that the difference with the set Exp_d^c of expressions in the programming language is that in logical expressions we can use logical variables, refer to the instance variables of other objects, and write conditional expressions. Furthermore, we extended the domain of discourse by means of logical variables ranging over sequences and notations to express the length of a sequence and the selection of an element from a sequence. The selection operator \cdot can be distinguished from the dereferencing operator $'$ by its higher vertical position on the line and by the type of its first argument.

Definition 4.4

The set Ass^c of assertions in class c , with typical elements P^c and Q^c , is defined by:

$$\begin{array}{ll}
 P^c ::= l_{\text{Bool}}^c & \\
 \quad | P^c \rightarrow Q^c & \\
 \quad | \neg P^c & \\
 \quad | \forall z_a P^c & \\
 \quad | \exists z_a P^c &
 \end{array}$$

This definition is rather conventional.

Definition 4.5

Of course, we shall freely use the logical connectives \wedge , \vee , and \leftrightarrow , which we consider as abbreviations of appropriate constructions with \rightarrow and \neg . Furthermore we shall use $l_d^c \uparrow$ as an abbreviation for $l_d^c \doteq \text{nil}_d$ and $l_d^c \downarrow$ for $\neg l_d^c \doteq \text{nil}_d$.

Definition 4.6

Finally we define the set $\text{Corr}F^c$ of *correctness formulae* in class c , with typical element F^c , as follows:

$$F^c ::= P^c \mid \{P^c\}\rho^c\{Q^c\}$$

4.2 Semantics of assertions and correctness formulae

Definition 4.7

In order to be able to assign a semantics to logical expressions we first define the set Ω of *valuations*, with typical element ω , as follows:

$$\Omega = \prod_a (LVar_a \rightarrow \mathbf{O}_\perp^a).$$

(Remember that $\mathbf{O}_\perp^a = \mathbf{O}^a$ if $a \in C^*$.) A valuation assigns a value to each logical variable.

Definition 4.8

We call a valuation ω *compatible* with a state σ if

- $\forall c \in C \forall z_c \in LVar_c \quad \omega_{(c)}(z) \in \sigma_\perp^{(c)}$
- $\forall c \in C \forall z_{c^*} \in LVar_{c^*} \forall n \in \mathbf{Z} \quad \text{elt}^c(\omega_{(c^*)}(z), n) \in \sigma_\perp^{(c)}$

Again an abbreviation is useful: we shall write $OK(\sigma, \delta, \omega)$ meaning that σ is consistent, δ agrees with σ , and ω is compatible with σ .

Lemma 4.9

Concerning the preservation of compatibility by statements and programs we have the following properties:

- For any statement S^c , for any agreement-preserving environment γ , for any state $\sigma \in \Sigma$, for any context δ^c and for any valuation ω such that $OK(\sigma, \delta, \omega)$ we have if $\sigma' = S[S](\gamma)(\delta)(\sigma)$ and $\sigma' \neq \perp$ then $OK(\sigma', \delta, \omega)$.

- For any program ρ^c , for any agreement-preserving environment γ , for any state $\sigma \in \Sigma$, for any context δ^c and for any valuation ω such that $OK(\sigma, \delta, \omega)$ we have if $\sigma' = \mathcal{P}[\rho](\gamma)(\delta)(\sigma)$ and $\sigma' \neq \perp$ then $OK(\sigma', \delta, \omega)$.

Proof

This is an easy consequence of lemma 3.21. □

Definition 4.10

We define the semantics of logical expressions by specifying the function

$$\mathcal{L}_a^c : LExpr_a^c \rightarrow \Omega \rightarrow \Delta^c \rightarrow \Sigma \rightarrow \mathbf{O}_\perp^a$$

as follows:

$$\mathcal{L}_d^c[e_d^c](\omega)(\delta)(\sigma) = \mathcal{E}_d^c[e](\delta)(\sigma)$$

$$\mathcal{L}_d^c[z_d](\omega)(\delta)(\sigma) = \omega_{(d)}(z)$$

$$\begin{aligned} \mathcal{L}_d^c[l_{c'}^c . x_{d'}^{c'}](\omega)(\delta)(\sigma) &= \perp && \text{if } \beta = \perp \\ &= \sigma_{(2)(c',d)}(\beta)(x_{d'}^{c'}) && \text{otherwise} \end{aligned}$$

$$\text{where } \beta^{c'} = \mathcal{L}_{c'}^c[l](\omega)(\delta)(\sigma)$$

$$\begin{aligned} \mathcal{L}_d^c[\text{if } l_{0_{\text{Bool}}}^c \text{ then } l_{1_d}^c \text{ else } l_{2_d}^c \text{ fi}](\omega)(\delta)(\sigma) &= \perp && \text{if } \beta = \perp \\ &= \mathcal{L}_d^c[l_1](\omega)(\delta)(\sigma) && \text{if } \beta = \mathbf{t} \\ &= \mathcal{L}_d^c[l_2](\omega)(\delta)(\sigma) && \text{if } \beta = \mathbf{f} \end{aligned}$$

$$\text{where } \beta = \mathcal{L}_d^c[l_0](\omega)(\delta)(\sigma)$$

$$\begin{aligned} \mathcal{L}_d^c[l_{1_{d'}}^c \dot{=} l_{2_{d'}}^c](\omega)(\delta)(\sigma) &= \mathbf{t} && \text{if } \mathcal{L}_{d'}^c[l_1](\omega)(\delta)(\sigma) = \mathcal{L}_{d'}^c[l_2](\omega)(\delta)(\sigma) \\ &= \mathbf{f} && \text{otherwise} \end{aligned}$$

(only if $d = \text{Bool}$)

$$\begin{aligned} \mathcal{L}_d^c[l_{1_d}^c + l_{2_d}^c](\omega)(\delta)(\sigma) &= \perp && \text{if } \mathcal{L}_d^c[l_1](\omega)(\delta)(\sigma) = \perp \\ &&& \text{or } \mathcal{L}_d^c[l_2](\omega)(\delta)(\sigma) = \perp \\ &= \mathcal{L}_d^c[l_1](\omega)(\delta)(\sigma) + \mathcal{L}_d^c[l_2](\omega)(\delta)(\sigma) && \text{otherwise} \end{aligned}$$

(only if $d = \text{Int}$)

⋮

$$\begin{aligned} \mathcal{L}_d^c[l_{1_{d'}}^c < l_{2_{d'}}^c](\omega)(\delta)(\sigma) &= \perp && \text{if } \mathcal{L}_{d'}^c[l_1](\omega)(\delta)(\sigma) = \perp \\ &&& \text{or } \mathcal{L}_{d'}^c[l_2](\omega)(\delta)(\sigma) = \perp \\ &= \mathbf{t} && \text{if } \mathcal{L}_{d'}^c[l_1](\omega)(\delta)(\sigma) < \mathcal{L}_{d'}^c[l_2](\omega)(\delta)(\sigma) \\ &= \mathbf{f} && \text{otherwise} \end{aligned}$$

(only if $d = \text{Bool}$ and $d' = \text{Int}$)

$$\begin{aligned}\mathcal{L}_{\text{Int}}^c[\![l_{d^*}]\!](\omega)(\delta)(\sigma) &= \text{len}^d(\mathcal{L}_{d^*}^c[\![l]\!](\omega)(\delta)(\sigma)) \\ \mathcal{L}_{d^*}^c[\![l_{1d^*} \cdot l_{2\text{Int}}]\!](\omega)(\delta)(\sigma) &= \text{elt}^d(\mathcal{L}_{d^*}^c[\![l_1]\!](\omega)(\delta)(\sigma), \mathcal{L}_{\text{Int}}^c[\![l_2]\!](\omega)(\delta)(\sigma))\end{aligned}$$

These equations are just what one would expect, especially after having seen definition 3.12.

Lemma 4.11

If $\sigma \in \Sigma$, $\delta \in \Delta^c$, and $\omega \in \Omega$ are such that $OK(\sigma, \delta, \omega)$, then for every logical expression $l \in LExpr_d^c$ we have $\mathcal{L}[\![l]\!](\omega)(\delta)(\sigma) \in \sigma_{\perp}^{(d)}$, and for every expression $l \in LExpr_{d^*}^c$ we have $\text{elt}^d(\mathcal{L}[\![l]\!](\omega)(\delta)(\sigma), n) \in \sigma_{\perp}^{(d)}$, for every n .

Proof

Induction on the complexity of l . □

Definition 4.12

Now we can define the semantics of assertions in terms of the function

$$\mathcal{A}^c : Ass^c \rightarrow \Omega \rightarrow \Delta^c \rightarrow \Sigma \rightarrow \mathbf{B}$$

as follows:

$$\begin{aligned}\mathcal{A}^c[\![l_{\text{Bool}}^c]\!](v)(\omega)(\delta)(\sigma) &= \mathbf{t} \quad \text{if } \mathcal{L}_{\text{Bool}}^c[\![l]\!](\omega)(\delta)(\sigma) = \mathbf{t} \\ &= \mathbf{f} \quad \text{otherwise} \\ \mathcal{A}^c[\![\neg P^c]\!](v)(\omega)(\delta)(\sigma) &= \mathbf{f} \quad \text{if } \mathcal{A}^c[\![P]\!](v)(\omega)(\delta)(\sigma) = \mathbf{t} \\ &= \mathbf{t} \quad \text{otherwise} \\ \mathcal{A}^c[\![\forall z_d P^c]\!](v)(\omega)(\delta)(\sigma) &= \mathbf{t} \quad \text{if for all } \beta \in \sigma^{(d)} \text{ we have} \\ &\quad \mathcal{A}^c[\![P^c]\!](v)(\omega\{\beta/z\})(\delta)(\sigma) = \mathbf{t} \\ &= \mathbf{f} \quad \text{otherwise} \\ \mathcal{A}^c[\![\exists z_d P^c]\!](v)(\omega)(\delta)(\sigma) &= \mathbf{t} \quad \text{if there is a } \beta \in \sigma^{(d)} \text{ such that} \\ &\quad \mathcal{A}^c[\![P^c]\!](v)(\omega\{\beta/z\})(\delta)(\sigma) = \mathbf{t} \\ &= \mathbf{f} \quad \text{otherwise} \\ \mathcal{A}^c[\![\forall z_{d^*} P^c]\!](\omega)(\delta)(\sigma) &= \mathbf{t} \quad \text{if for all } \alpha \in \mathbf{O}^{d^*} \text{ such that} \\ &\quad \forall n \in \mathbf{Z} \text{ elt}(\alpha, n) \in \sigma_{\perp}^{(d)} \text{ we have} \\ &\quad \mathcal{A}^c[\![P^c]\!](\omega\{\alpha/z\})(\delta)(\sigma) = \mathbf{t} \\ &= \mathbf{f} \quad \text{otherwise} \\ \mathcal{A}^c[\![\exists z_{d^*} P^c]\!](\omega)(\delta)(\sigma) &= \mathbf{t} \quad \text{if there is an } \alpha \in \mathbf{O}^{d^*} \text{ such that} \\ &\quad \forall n \in \mathbf{Z} \text{ elt}(\alpha, n) \in \sigma_{\perp}^{(d)} \text{ and} \\ &\quad \mathcal{A}^c[\![P^c]\!](\omega\{\alpha/z\})(\delta)(\sigma) = \mathbf{t} \\ &= \mathbf{f} \quad \text{otherwise}\end{aligned}$$

A few remarks should be made here.

- Note that the possible values of a boolean logical expression are **t**, **f**, and \perp . If such an expression is viewed as an assertion, only **t** and **f** remain. If viewed as an expression it yields \perp , as an assertion it delivers **f**.
- It is very important to note that in assertions of the form $\forall z_d P$ and $\exists z_d P$ the quantification ranges only over the *existing*, non-nil objects of the appropriate type. In assertions of the form $\forall z_a P$ and $\exists z_a P$ (where $a = d^*$, for some $d \in C$) the quantification ranges over sequences of existing objects, possibly nil.

Example 4.13

The formula

$$v \xrightarrow{x} w$$

from [7] can be expressed in our new assertion language in the following way:

$$\exists z_{d^*} (z \cdot 1 \doteq v \wedge z \cdot |z| \doteq w \wedge \forall n (0 < n \wedge n < |z|) \rightarrow (z \cdot n).x \doteq z \cdot (n+1))$$

Here n denotes a logical variable ranging over integers. This formula expresses that the object w can be reached from v by a “ x -path”.

Example 4.14

There are no logical expressions in the language to construct a sequence with one specific element (a singleton). However, if we want to say that property P holds for the singleton whose element is given by the logical expression l , we can do this as follows:

$$\exists z_{d^*} |z| \doteq 1 \wedge z \cdot 1 \doteq l \wedge P(z)$$

or equivalently:

$$\forall z_{d^*} (|z| \doteq 1 \wedge z \cdot 1 \doteq l) \rightarrow P(z).$$

A similar procedure is possible for the empty sequence and for the concatenation of two sequences. Furthermore we can see whether two sequences are equal by checking if they have the same lengths and whether their corresponding elements are equal. (Direct ways of expressing the above things could be included in the assertion language, but they would make the substitution operation $[new/u]$ in definitions 5.13 and 5.15 much more complicated.)

Definition 4.15

Finally we define the notion of *truth* and *validity* of correctness formulae.

- We say that a correctness formula of the form P^c is *true* with respect to a valuation ω , a context δ^c , and a state σ , written as $\omega, \delta, \sigma \models P$, if $OK(\sigma, \delta, \omega)$ and

$$\mathcal{A}^c[P](\omega)(\delta)(\sigma) = \mathbf{t}$$

- We call a correctness formula of the form P^c *valid*, written as $\models P$, if it is true with respect to every ω , δ^c , and σ such that $OK(\sigma, \delta, \omega)$.
- A correctness formula of the form $\{P^c\}\rho^c\{Q^c\}$ is called true with respect to an environment γ , a valuation ω , a context δ^c , and a state σ , written as $\gamma, \omega, \delta, \sigma \models \{P\}\rho\{Q\}$, if $\omega, \delta, \sigma \models P$ implies that for the state $\sigma' = \mathcal{P}^c[\rho](\gamma)(\delta)(\sigma)$ we have

$$\sigma' \neq \perp \Rightarrow \omega, \delta, \sigma' \models Q.$$

- We define a correctness formula of the form $\{P^c\}\rho^c\{Q^c\}$ to be valid with respect to an environment γ , written as $\gamma \models \{P\}\rho\{Q\}$, if we have $\gamma, \omega, \delta, \sigma \models \{P\}\rho\{Q\}$ for every ω , δ^c , and σ . We call such a correctness formula simply valid if it is valid with respect to every environment.

5 The proof system

In this section we shall present a number of axioms and rules that can be used to derive correctness formulae. For each axiom and rule we shall give its justification by proving that it is valid. Note that axioms are correctness formulae so we have already defined what validity means for them. We call a proof rule valid if for every environment γ the validity of the premisses of the rule with respect to γ implies the validity of the conclusion with respect to γ . The consequence of the validity of all the axioms and rules will be that our proof system is *sound*, i.e., that if we can derive a correctness formula (without any further assumptions), this correctness formula will be valid. (There is one rule in the system that cannot be proved valid in isolation: the recursion rule (MR) in definition 5.33. It will get a special treatment in the soundness proof of the whole proof system (see theorem 5.40).)

5.1 Simple assignments

Definition 5.1

We shall call a statement a *simple assignment* if it is of the form $x \leftarrow e$ or $u \leftarrow e$ (that is, it uses the first form of a side effect expression: the one without a side effect).

5.1.1 Simple assignment to a temporary variable

Definition 5.2

Our first axiom deals with the case that the target variable is a temporary variable:

$$\{P^c[e_d^c/u_d]\} \langle U|c : u_d \leftarrow e_d^c \rangle \{P^c\} \quad (\text{SAT})$$

Here the notation $P[e/u]$ means: P in which e is substituted for x . We shall formalize that notion in the next definition. (Note that this definition merely asserts that the name (SAT) refers to the class of formulae of the form listed above.)

Definition 5.3

We shall define the substitution operation $[e/u]$ first for logical expressions:

$$\begin{aligned} x \quad [e/u] &= x \\ u \quad [e/u] &= e \\ u' \quad [e/u] &= u' && \text{if } u' \neq u \\ z \quad [e/u] &= z \\ l \quad [e/u] &= l && \text{if } l = \text{nil, self, true, false} \\ n \quad [e/u] &= n \\ l.x \quad [e/u] &= (l[e/u]).x \end{aligned}$$

$$\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi} [e/u] = \text{if } l_0[e/u] \text{ then } l_1[e/u] \text{ else } l_2[e/u] \text{ fi}$$

$$(l_1 \doteq l_2)[e/u] = (l_1[e/u]) \doteq (l_2[e/u])$$

$$(l_1 + l_2)[e/u] = (l_1[e/u]) + (l_2[e/u])$$

$$\vdots$$

$$(l_1 < l_2)[e/u] = (l_1[e/u]) < (l_2[e/u])$$

$$\vdots$$

$$|l| \quad [e/u] = |l[e/u]|$$

$$(l_1 \cdot l_2)[e/u] = (l_1[e/u]) \cdot l_2[e/u]$$

Now we define this substitution for assertions other than logical expressions:

$$(P \rightarrow Q)[e/u] = (P[e/u]) \rightarrow (Q[e/u])$$

$$(\neg P) \quad [e/u] = \neg(P[e/u])$$

$$(\forall z \, P) \quad [e/u] = \forall z \, (P[e/u])$$

$$(\exists z \, P) \quad [e/u] = \exists z \, (P[e/u])$$

This definition can be summarized by saying that we can perform the substitution $[e/u]$ by replacing u by e everywhere in the expression or assertion at hand. However, this will not be true for the notions of substitution that we will define in the sequel, despite the fact that we use a very similar notation to indicate those substitutions.

In the following lemma we express the most important characteristic of the substitution $[e/u]$. Informally spoken, for any logical expression or assertion, the substituted form has the same value in the state *before* the assignment as the unsubstituted form has in the state *after* the assignment.

Lemma 5.4

Consider the assignment statement $u_d \leftarrow e_d^c$. Let $\gamma \in \Gamma$, $\sigma \in \Sigma$ and $\delta \in \Delta^c$ be arbitrary, and let

$$\sigma' = S[u \leftarrow e](\gamma)(\delta)(\sigma).$$

Then we have the following facts:

1. For every logical expression l_d^c and every valuation ω

$$\mathcal{L}[l[e/u]](\omega)(\delta)(\sigma) = \mathcal{L}[l](\omega)(\delta)(\sigma').$$

2. For every assertion P^c , every valuation ω

$$\mathcal{A}[P[e/u]](\omega)(\delta)(\sigma) = \mathcal{A}[P](\omega)(\delta)(\sigma').$$

Proof

First we observe that $\sigma' = S[u \leftarrow e](\gamma)(\delta)(\sigma)$ means that $\sigma' = \sigma\{\beta/u\}$, where $\beta = \mathcal{E}[e](\delta)(\sigma)$.

Now we can prove the first part of the lemma by induction with respect to the structure of l . The only interesting case occurs when $l = u$ so that $l[e/u] = e$:

$$\begin{aligned} \mathcal{L}[e](\omega)(\delta)(\sigma) &= \mathcal{E}[e](\delta)(\sigma) \\ &= \beta \\ &= \sigma'_{(3)(d)}(u) \\ &= \mathcal{E}[u](\delta)(\sigma') \\ &= \mathcal{L}[u](\omega)(\delta)(\sigma') \end{aligned}$$

After that we can prove the second part of the lemma by a straightforward induction on the structure of P .

Of course, this lemma is easily extended to the case where instead of an assignment *statement* we take a *program* in which the statement is a simple assignment to a temporary variable:

Corollary 5.5

The axiom (SAT) is valid, that is, for every environment γ we have

$$\gamma \models \{P[e/u]\} \langle U|c : u \leftarrow e \rangle \{P\}.$$

□

Note that the corollary uses only one direction of the lemma. The two directions together say that $P[e/u]$ is the *weakest precondition* of the statement $u \leftarrow e$ with respect to the postcondition P .

5.1.2 Simple assignment to an instance variable**Definition 5.6**

In the case that the target variable of an assignment statement is an instance variable, we use the following axiom:

$$\{P^c[e_d^c/x_d^c]\} \langle U|c : x_d^c \leftarrow e_d^c \rangle \{P^c\} \quad (\text{SAI})$$

This looks very similar to our first axiom (SAT), but note that we have not yet defined what substitution means if we substitute an expression for an instance variable instead of a temporary variable. We shall do that now, and the difference will become clear immediately:

Definition 5.7

The substitution operation $[e/x]$ is defined as follows on logical expressions:

$$\begin{aligned} x \quad [e/x] &= e \\ x' \quad [e/x] &= x' && \text{if } x' \neq x \\ u \quad [e/x] &= u \\ z \quad [e/x] &= z \\ l \quad [e/x] &= l && \text{if } l = \text{nil, self, true, false} \\ n \quad [e/x] &= n \\ l.x \quad [e/x] &= \text{if } (l[e/x]) \doteq \text{self then } e \text{ else } (l[e/x]).x \text{ fi} \\ l.x' \quad [e/x] &= (l[e/x]).x' && \text{if } x' \neq x \end{aligned}$$

$$\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi} [e/x] = \text{if } l_0[e/x] \text{ then } l_1[e/x] \text{ else } l_2[e/x] \text{ fi}$$

$$\begin{aligned}
(l_1 \doteq l_2)[e/x] &= (l_1[e/x] \doteq l_2[e/x]) \\
(l_1 + l_2)[e/x] &= (l_1[e/x] + l_2[e/x]) \\
&\vdots \\
(l_1 < l_2)[e/x] &= (l_1[e/x] < l_2[e/x]) \\
&\vdots \\
|l| [e/x] &= |l[e/x]| \\
(l_1 \cdot l_2)[e/x] &= (l_1[e/x] \cdot l_2[e/x])
\end{aligned}$$

The definition is extended to assertions other than logical expressions in the same way as before:

$$\begin{aligned}
(P \rightarrow Q)[e/x] &= (P[e/x] \rightarrow Q[e/x]) \\
(\neg P) [e/x] &= \neg(P[e/x]) \\
(\forall z P) [e/x] &= \forall z (P[e/x]) \\
(\exists z P) [e/x] &= \exists z (P[e/x])
\end{aligned}$$

The most important aspect of this definition is certainly the conditional expression that turns up when we are dealing with a logical expression of the form $l.x$. This is necessary because a certain form of *aliasing* is possible: the situation that different expressions refer to the same variable. In the case of $l.x$, it is possible that, after substitution, l refers to the currently active object, so that $l.x$ is the same variable as x and should be substituted by e . It is also possible that, after substitution, l does not refer to the currently executing object, and in this case no substitution should take place. Since we cannot decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides “dynamically”.

Lemma 5.8

Consider the assignment statement $x_d^c \leftarrow e_d^c$. Let $\gamma \in \Gamma$, $\sigma \in \Sigma$, and $\delta \in \Delta^c$ be arbitrary, and let

$$\sigma' = S[x \leftarrow e](\gamma)(\delta)(\sigma).$$

Then we have the following facts:

1. For every logical expression l_d^c , and every valuation ω

$$\mathcal{L}[l[e/x]](\omega)(\delta)(\sigma) = \mathcal{L}[l](\omega)(\delta)(\sigma').$$

2. For every assertion P^c and every valuation ω

$$\mathcal{A}[P[e/x]](\omega)(\delta)(\sigma) = \mathcal{A}[P](\omega)(\delta)(\sigma').$$

Proof

Like in lemma 5.4 we first note that $\sigma' = \sigma\{\beta/\delta_{(1)}, x\}$ where $\beta = \mathcal{E}[e](\delta)(\sigma)$. The first part of the lemma is now proved by induction on the complexity of l . We shall only deal with the most interesting case: $l = l' . x$. The induction hypothesis tells us that $\mathcal{L}[l'[e/x]](\omega)(\delta)(\sigma) = \mathcal{L}[l'](\omega)(\delta)(\sigma')$. Let us call this object β_0 . Now if $\beta_0 = \delta_{(1)}$ then $\mathcal{L}[l' . x](\omega)(\delta)(\sigma') = \sigma'_{(2)}(\delta_{(1)})(x) = \beta = \mathcal{L}[e](\omega)(\delta)(\sigma)$. Otherwise we have $\mathcal{L}[l' . x](\omega)(\delta)(\sigma') = \sigma'_{(2)}(\beta_0)(x) = \mathcal{L}[(l'[e/x]) . x](\omega)(\delta)(\sigma)$. So $\mathcal{L}[\text{if } l'[e/x] \doteq \text{self then } e \text{ else } (l'[e/x]) . x \text{ fi}](\omega)(\delta)(\sigma) = \mathcal{L}[l' . x](\omega)(\delta)(\sigma')$.

The rest of the lemma is proved in a way similar to lemma 5.4. □

Again we can extend this to programs instead of statements:

Corollary 5.9

The axiom (SAI) is valid, that is, for every environment γ we have

$$\gamma \models \{P[e/x]\} \langle U|c : x \leftarrow e \rangle \{P\}.$$

□

Note that this corollary also uses only one direction of the corresponding lemma. Again the two directions together say that $P[e/x]$ is the weakest precondition of the statement $x \leftarrow e$ with respect to the postcondition P .

5.2 Creating new objects

5.2.1 Assigning a new object to a temporary variable

Definition 5.10

For an assignment of the form $u \leftarrow \text{new}$ we have a axiom similar to the previous two:

$$\{P^c[\text{new}_{c'}/u_{c'}]\} \langle U|c : u_{c'} \leftarrow \text{new}_{c'} \rangle \{P^c\} \quad (\text{NT})$$

Again we still have to define what this notion of substitution looks like, but first we shall define the substitution of an expression for a *logical* variable, because we shall need that later:

Definition 5.11

We define the substitution operation $[e/z]$ on logical expressions by:

$$\begin{aligned}
x \quad [e/z] &= x \\
u \quad [e/z] &= u \\
z \quad [e/z] &= e \\
z' \quad [e/z] &= z' && \text{if } z' \neq z \\
l' \quad [e/z] &= l' && \text{if } l' = \text{nil, self, true, false} \\
n \quad [e/z] &= n \\
l' . x \quad [e/z] &= (l'[e/z]) . x \\
\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi} [e/z] &= \text{if } l_0[e/z] \text{ then } l_1[e/z] \text{ else } l_2[e/z] \text{ fi} \\
(l_1 \doteq l_2)[e/z] &= (l_1[e/z]) \doteq (l_2[e/z]) \\
(l_1 + l_2)[e/z] &= (l_1[e/z]) + (l_2[e/z]) \\
&\vdots \\
(l_1 < l_2)[e/z] &= (l_1[e/z]) < (l_2[e/z]) \\
&\vdots \\
|l| \quad [e/z] &= |l[e/z]| \\
(l_1 \cdot l_2)[e/z] &= (l_1[e/z]) \cdot l_2[e/z]
\end{aligned}$$

We extend this definition to assertions other than logical expressions as follows:

$$\begin{aligned}
(P \rightarrow Q)[e/z] &= (P[e/z]) \rightarrow (Q[e/z]) \\
(\neg P) \quad [e/z] &= \neg(P[e/z]) \\
(\forall z \, P) \quad [e/z] &= \forall z \, P \\
(\forall z' \, P) \quad [e/z] &= \forall z' (P[e/z]) && \text{if } z' \neq z \\
(\exists z \, P) \quad [e/z] &= \exists z \, P \\
(\exists z' \, P) \quad [e/z] &= \exists z' (P[e/z]) && \text{if } z' \neq z
\end{aligned}$$

This definition can be summarized by observing that the substitution can be carried out by replacing z by e everywhere except in the scope of a quantifier where z is bound.

Lemma 5.12

Let $\sigma \in \Sigma$, $\delta \in \Delta^c$, $e \in \text{Exp}_d^c$, and $z \in \text{LVar}_d$ be arbitrary, and let $\beta = \mathcal{E}[e](\delta)(\sigma)$. Then we have

1. For all $l \in \text{LExp}_d^c$, and for all $\omega \in \Omega$:

$$\mathcal{L}[l[e/z]](\omega)(\delta)(\sigma) = \mathcal{L}[l](\omega\{\beta/z\})(\delta)(\sigma).$$

2. For all $P \in Ass^c$, for all $\omega \in \Omega$:

$$\mathcal{A}[P[e/z]](\omega)(\delta)(\sigma) = \mathcal{A}[P](\omega\{\beta/z\})(\delta)(\sigma).$$

Proof

A rather trivial induction on the complexity of l and P . □

Now we can define the substitution $[new_c/u_c]$. We shall do this first for logical expressions. As with the notions of substitution used in the axioms for simple assignments, we want the expression after substitution to have the same meaning in a state before the assignment as the unsubstituted expression has in the state after the assignment. However, in the case of a new-assignment, there are expressions for which this is not possible, because they refer to the new object (in the new state) and there is no expression that could refer to that object in the old state, because it does not exist yet. Therefore the result of the substitution must be left undefined in some cases.

However we will show that we *are* able to carry out the substitution. The idea behind this is that in such an assertion the variable u referring to the new object can essentially occur only in a context where either one of its instance variables is referenced, or it is compared for equality with another expression. In both of these cases we can predict the outcome without having to refer to the new object.

Definition 5.13

Here comes the formal definition of the substitution $[new/u]$ for logical expressions:

$$\begin{aligned} x [new/u] &= x \\ u [new/u] &\text{ is undefined} \\ u' [new/u] &= u \quad \text{if } u' \neq u \\ z [new/u] &= z \\ l [new/u] &= l \quad \text{if } l = \text{nil, self, true, false} \\ n [new/u] &= n \\ \\ x' . x [new/u] &= x' . x \\ u . x [new/u] &= \text{nil} \\ u' . x [new/u] &= u' . x \quad \text{if } u' \neq u \\ z . x [new/u] &= z . x \\ l . x [new/u] &= l . x \quad \text{if } l = \text{nil, self} \\ l . x' . x [new/u] &= (l . x' [new/u]) . x \\ \\ (\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi } . x) [new/u] \\ &= \text{if } l_0 [new/u] \text{ then } (l_1 . x) [new/u] \text{ else } (l_2 . x) [new/u] \text{ fi} \end{aligned}$$

if l_0 then l_1 else l_2 fi[new/ u]
 = if l_0 [new/ u] then l_1 [new/ u] else l_2 [new/ u] fi
 if the substitutions of the subexpressions are all defined,
 otherwise undefined

$(l_1 \doteq l_2)$ [new/ u] = $(l_1$ [new/ u]) \doteq $(l_2$ [new/ u])
 if neither l_1 nor l_2 is u or of the form if ... fi

$(l_1 \doteq l_2)$ [new/ u] = false
 if either $l_1 = u$ and l_2 is not u or of the form if ... fi
 or $l_2 = u$ and l_1 is not u or of the form if ... fi

$(l_1 \doteq l_2)$ [new/ u] = true
 if $l_1 = l_2 = u$

$(\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi} \doteq l_3)$ [new/ u]
 = if l_0 [new/ u] \uparrow
 then $(l_3 \uparrow)$ [new/ u]
 else if l_0 [new/ u]
 then $(l_1 \doteq l_3)$ [new/ u]
 else $(l_2 \doteq l_3)$ [new/ u]
 fi
 fi

$(l_1 \doteq \text{if } l_0 \text{ then } l_2 \text{ else } l_3 \text{ fi})$ [new/ u]
 = if l_0 [new/ u] \uparrow
 then $(l_1 \uparrow)$ [new/ u]
 else if l_0 [new/ u]
 then $(l_1 \doteq l_2)$ [new/ u]
 else $(l_1 \doteq l_3)$ [new/ u]
 fi
 fi
 if l_1 is not of the form if ... fi

$$\begin{aligned}
(l_1 + l_2)[\text{new}/u] &= (l_1[\text{new}/u]) + (l_2[\text{new}/u]) \\
&\vdots \\
(l_1 < l_2)[\text{new}/u] &= (l_1[\text{new}/u]) < (l_2[\text{new}/u]) \\
&\vdots \\
|l|[\text{new}/u] &= |l[\text{new}/u]| \\
(l_1 \cdot l_2)[\text{new}/u] &= (l_1[\text{new}/u]) \cdot (l_2[\text{new}/u])
\end{aligned}$$

Lemma 5.14

Let $u \in TVar_d$ with $d \in C$ (i.e., $d \neq \text{Int}, \text{Bool}$).

1. For every logical expression l we have that $l[\text{new}/u]$ is defined if and only if l is *not* of the form indicated by the following BNF definition:

$$\begin{aligned}
lu &::= u \\
&| \text{ if } l_0 \text{ then } lu \text{ else } l_1 \text{ fi} \\
&| \text{ if } l_0 \text{ then } l_1 \text{ else } lu \text{ fi}
\end{aligned}$$

2. If $\sigma \in \Sigma$, $\delta \in \Delta^c$, $\omega \in \Omega$, and $\gamma \in \Gamma$ are such that $OK(\sigma, \delta, \omega)$, and if $\sigma' = S[u \leftarrow \text{new}](\gamma)(\delta)(\sigma)$ then for every logical expression l such that $l[\text{new}/u]$ is defined we have

$$\mathcal{L}[l[\text{new}/u]](\omega)(\delta)(\sigma) = \mathcal{L}[l](\omega)(\delta)(\sigma').$$

Proof

The first part is easily proved by induction on the complexity of l . For the second part we first observe that

$$\sigma' = \sigma \{ \sigma_{(1)(d)} \cup \{\beta\} / d \} \{ \beta / u \}$$

where $\beta = \text{pick}^d(\sigma_{(1)(d)})$, so $\beta \notin \sigma_{(1)(d)} \cup \{\perp\}$ (see definitions 3.13 and 3.14).

Now we can prove our lemma by induction on the complexity of l . In several places we need the information that $OK(\sigma, \delta, \omega)$ together with lemma 4.11 in order to prove that the result of an intermediate logical expression is not equal to β . Let us deal with one representative case: $l = x' . x$. Then $l[\text{new}/u] = l = x' . x$. Now the induction hypothesis tells us that $\mathcal{L}[x'](\omega)(\delta)(\sigma) = \mathcal{L}[x'](\omega)(\delta)(\sigma')$. If we put this equal to β' then we know $\beta' \neq \beta$ because lemma 4.11 tells us that $\beta' \in \sigma_{(1)(d)} \cup \{\perp\}$. Therefore we have $\mathcal{L}[x' . x](\omega)(\delta)(\sigma) = \sigma_{(2)}(\beta')(x) = \sigma'_{(2)}(\beta')(x) = \mathcal{L}[x' . x](\omega)(\delta)(\sigma')$. \square

Definition 5.15

We extend the substitution operation $[new/u]$ to assertions other than logical expressions as follows (we assume that the type of u is $d \in C$):

$$\begin{aligned}
(P \rightarrow Q)[new/u] &= (P[new/u]) \rightarrow (Q[new/u]) \\
(\neg P)[new/u] &= \neg(P[new/u]) \\
(\forall z_d P)[new/u] &= (\forall z(P[new/u])) \wedge (P[u/z][new/u]) \\
(\forall z_{d^*} P)[new/u] &= (\forall z \forall z'_{Bool^*} |z| \doteq |z'| \rightarrow (P[z', u/z][new/u])) \\
(\forall z_a P)[new/u] &= (\forall z(P[new/u])) \quad \text{if } a \neq d, d^* \\
(\exists z_d P)[new/u] &= (\exists z(P[new/u])) \vee (P[u/z][new/u]) \\
(\exists z_{d^*} P)[new/u] &= (\exists z \exists z'_{Bool^*} |z| \doteq |z'| \wedge (P[z', u/z][new/u])) \\
(\exists z_a P)[new/u] &= (\exists z(P[new/u])) \quad \text{if } a \neq d, d^*
\end{aligned}$$

Here we choose for z' the first variable from $LVar_{Bool^*}$ that does not occur in P . The idea is that z and z' together code a sequence of objects in the state after the new-statement. At the places where z' yields **t** the value of the coded sequence is the newly created object. Where z' yields **f** the value of the coded sequence is the same as the value of z and where z' delivers \perp the coded sequences also yields \perp .

We still have to define the substitution operation $[z', u/z]$ and we shall do that now:

Definition 5.16

Let $d \in C$, $u \in TVar_d$, $z \in LVar_{d^*}$, and $z' \in LVar_{Bool^*}$. For logical expressions we define the operation $[z', u/z]$ as follows:

$$\begin{aligned}
e \quad [z', u/z] &= e \\
z \quad [z', u/z] &\text{ is undefined} \\
z'' \quad [z', u/z] &= z'' \quad \text{if } z'' \neq z \\
l.x \quad [z', u/z] &= (l[z', u/z]).x \\
|z| \quad [z', u/z] &= |z| \\
|l| \quad [z', u/z] &= |l[z', u/z]| \quad \text{if } l \neq z \\
(z \cdot l_2)[z', u/z] &= \text{if } z' \cdot (l_2[z', u/z]) \text{ then } u \text{ else } z \cdot (l_2[z', u/z]) \text{ fi} \\
(l_1 \cdot l_2)[z', u/z] &= (l_1[z', u/z]) \cdot (l_2[z', u/z]) \quad \text{if } l_1 \neq z
\end{aligned}$$

$$\begin{aligned}
&\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi } [z', u/z] = \\
&\quad \text{if } (l_0[z', u/z]) \text{ then } (l_1[z', u/z]) \text{ else } (l_2[z', u/z]) \text{ fi}
\end{aligned}$$

$$\begin{aligned}
(l_1 \doteq l_2)[z', u/z] &= (l_1[z', u/z]) \doteq (l_2[z', u/z]) \\
(l_1 + l_2)[z', u/z] &= (l_1[z', u/z]) + (l_2[z', u/z]) \\
&\vdots \\
(l_1 < l_2)[z', u/z] &= (l_1[z', u/z]) < (l_2[z', u/z]) \\
&\vdots \\
|l|[z', u/z] &= |l[z', u/z]| \\
(l_1 \cdot l_2)[z', u/z] &= (l_1[z', u/z]) \cdot l_2[z', u/z]
\end{aligned}$$

We extend this definition to assertions other than logical expressions as follows:

$$\begin{aligned}
(P \rightarrow Q)[z', u/z] &= (P[z', u/z]) \rightarrow (Q[z', u/z]) \\
(\neg P)[z', u/z] &= \neg(P[z', u/z]) \\
(\forall z P)[z', u/z] &= (\forall z P) \\
(\forall z'' P)[z', u/z] &= (\forall z'' (P[z', u/z])) && \text{if } z'' \neq z \\
(\exists z P)[z', u/z] &= (\exists z P) \\
(\exists z'' P)[z', u/z] &= (\exists z'' (P[z', u/z])) && \text{if } z'' \neq z
\end{aligned}$$

Lemma 5.17

Let u, z, z' be as in definition 5.16. Let $\sigma \in \Sigma$, $\delta \in \Delta^c$, $\omega \in \Omega$, and take $\alpha = \omega_{(d^*)}(z)$, $\alpha' = \omega_{(\text{Bool}^*)}(z')$, $\beta = \sigma_{(3)(d)}(u)$. Suppose that $\text{len}^d(\alpha) = \text{len}^{\text{Bool}}(\alpha')$. Define $\alpha'' \in \mathbf{O}^{d^*}$ to be the sequence that satisfies (for all $n \in \mathbf{Z}$):

$$\begin{aligned}
\text{len}(\alpha'') &= \text{len}(\alpha) \\
\text{elt}(\alpha'', n) &= \beta && \text{if } \text{elt}(\alpha', n) = \mathbf{t} \\
\text{elt}(\alpha'', n) &= \text{elt}(\alpha, n) && \text{if } \text{elt}(\alpha', n) = \mathbf{f} \\
\text{elt}(\alpha'', n) &= \perp && \text{if } \text{elt}(\alpha', n) = \perp
\end{aligned}$$

and take $\omega' = \omega\{\alpha''/z\}$.

Then we have:

1. For every $l \in \text{LEXP}_a^c$ such that $l \neq z$:

$$\mathcal{L}_a^c[l[z', u/z]](\omega)(\delta)(\sigma) = \mathcal{L}_a^c[l](\omega')(\delta)(\sigma).$$

2. For every $P \in \text{Ass}^c$ such that z' does not occur in P :

$$\mathcal{A}^c[P[z', u/z]](\omega)(\delta)(\sigma) = \mathcal{A}^c[P](\omega')(\delta)(\sigma).$$

Proof

The proof consists of a quite easy induction on the complexity of l and P respectively. Of course, the only interesting case is when l is of the form $z \cdot l_2$. Note that the condition on z' is necessary to exclude assertions of the form $\forall z' P$ or $\exists z' P$. \square

Lemma 5.18

Let $\sigma \in \Sigma$, $\delta \in \Delta^c$, $\omega \in \Omega$ such that $OK(\sigma, \delta, \omega)$. Let $d \in C$, $u \in TVar_d$, $\gamma \in \Gamma$ and define $\sigma' = S^c[u \leftarrow \text{new}](\gamma)(\delta)(\sigma)$. Then for every assertion $P \in Ass^c$ we have

$$\mathcal{A}^c[P[\text{new}/u]](\omega)(\delta)(\sigma) = \mathcal{A}^c[P](\omega)(\delta)(\sigma').$$

Proof

Again we use induction on the complexity of P . The only case which is not yet clear from the first approach is quantification over sequences, so let us consider the case where $P = \forall z_{d^*} Q$. Take $\beta = f^d(\sigma^{(d)})$, so that $\sigma'^{(d)} = \sigma^{(d)} \cup \{\beta\}$ and $\beta = \sigma'_{(3)(d)}(u)$, and let z' be the first variable from $LVar_{Bool^*}$ that does not occur in Q .

Now suppose that

$$\mathcal{A}[(\forall z_{d^*} Q)[\text{new}/u]](\omega)(\delta)(\sigma) = t.$$

We shall prove that

$$\mathcal{A}[\forall z_{d^*} Q](\omega)(\delta)(\sigma') = t$$

so we have to show that for every $\alpha'' \in \mathbf{O}^{d^*}$ such that $\text{elt}(\alpha'', n) \in \sigma'^{(d)}_{\perp}$ for all $n \in \mathbb{Z}$, it is the case that $\mathcal{A}[Q](\omega\{\alpha''/z\})(\delta)(\sigma') = t$. If we have such an α'' , we can define $\alpha \in \mathbf{O}^{d^*}$ and $\alpha' \in \mathbf{O}^{Bool^*}$ as follows:

$$\begin{aligned} \text{len}(\alpha) &= \text{len}(\alpha') = \text{len}(\alpha'') \\ \text{elt}(\alpha, n) &= \perp, & \text{elt}(\alpha', n) &= t \quad \text{if } \text{elt}(\alpha'', n) = \beta \\ \text{elt}(\alpha, n) &= \text{elt}(\alpha'', n), & \text{elt}(\alpha', n) &= f \quad \text{if } 1 \leq n \leq \text{len}(\alpha'') \\ & & & \text{and } \text{elt}(\alpha'', n) \neq \beta \end{aligned}$$

Now because

$$\mathcal{A}[\forall z_{d^*} \forall z'_{Bool^*} |z| \doteq |z'| \rightarrow Q[z', u/z][\text{new}/u]](\omega)(\delta)(\sigma) = t$$

and because α and α' have equal length and do not have elements outside $\sigma^{(d)}_{\perp}$ and $\sigma^{(Bool)}_{\perp}$ respectively, we know that

$$\mathcal{A}[Q[z', u/z][\text{new}/u]](\omega\{\alpha/z\}\{\alpha'/z'\})(\delta)(\sigma) = t.$$

The induction hypothesis then tells us that

$$\mathcal{A}[Q[z', u/z]](\omega\{\alpha/z\}\{\alpha'/z'\})(\delta)(\sigma') = t.$$

Finally we can apply lemma 5.17 and use the fact that z' does not occur in Q , to see that

$$\mathcal{A}[Q](\omega\{\alpha''/z\})(\delta)(\sigma') = \mathbf{t}.$$

To prove that $\mathcal{A}[\forall z_d. Q](\omega)(\delta)(\sigma') = \mathbf{t}$ implies $\mathcal{A}[(\forall z_d. Q)[\text{new}/u]](\omega)(\delta)(\sigma) = \mathbf{t}$ involves reasoning in the other direction, in particular to find a suitable α'' for each pair α, α' that satisfies certain conditions. We omit further details. \square

Again we extend this result to the case of programs:

Corollary 5.19

The axiom (NT) is valid, that is, for every environment γ we have

$$\gamma \models \{P[\text{new}/u]\} \langle U|c : u \leftarrow \text{new} \rangle \{P\}.$$

\square

5.2.2 Assigning a new object to an instance variable

Definition 5.20

If our assignment is of the form $x \leftarrow \text{new}$ we have the following axiom:

$$\{P^c[\text{new}_{c'}/x_{c'}^c]\} \langle U|c : x_{c'}^c \leftarrow \text{new}_{c'} \rangle \{P^c\} \quad (\text{NI})$$

Fortunately, after having worked through the previous subsection, this new axiom is simple to define and to prove valid.

Definition 5.21

The substitution operation $[\text{new}_{c'}/x_{c'}^c]$ is defined by:

$$P[\text{new}_{c'}/x_{c'}^c] = P[u_{c'}/x_{c'}^c] [\text{new}_{c'}/u_{c'}]$$

where $u_{c'}$ is a temporary variable that does not occur in P . (It is easy to see that this definition does not depend on the actual u used.)

Lemma 5.22

Let $\sigma \in \Sigma$, $\delta \in \Delta^c$ and $\omega \in \Omega$ be such that $OK(\sigma, \delta, \omega)$. Let $\gamma \in \Gamma$, $d \in C$, $x \in IVar_d^c$, and define $\sigma' = \mathcal{S}[x \leftarrow \text{new}](\gamma)(\delta)(\sigma)$. Then for every assertion P^c we have

$$\omega, \delta, \sigma \models P[\text{new}/x] \iff \omega, \delta, \sigma' \models P.$$

Proof

Choose some $u \in TVar_d$ which does not occur in P , so that we have $P[\text{new}/x] =$

$P[u/x][\text{new}/u]$. Let $\sigma'' = \mathcal{S}[u \leftarrow \text{new}; x \leftarrow u](\gamma)(\delta)(\sigma)$. We have by lemma 5.8 and lemma 5.18 that $\omega, \delta, \sigma \models P[u/x][\text{new}/u] \iff \omega, \delta, \sigma'' \models P$.

Now if $\beta = \text{pick}^d(\sigma^{(d)})$ then we have $\sigma' = \sigma\{\beta/\delta_{(1)}, x\}$ and $\sigma'' = \sigma\{\beta/u\}\{\beta/\delta_{(1)}, x\}$, so that $\sigma'' = \sigma'\{\beta/u\}$. Because u does not occur in P we have $\omega, \delta, \sigma' \models P \iff \omega, \delta, \sigma'' \models P$, and the result of the lemma follows. \square

Corollary 5.23

The axiom (NI) is valid, that is, for every environment γ we have

$$\gamma \models \{P[\text{new}/x]\} \langle U | c : x \leftarrow \text{new} \rangle \{P\}.$$

\square

5.3 Sending messages

In this subsection we present some proof rules for verifying the third kind of assignments: the ones where a message is sent and the result stored in the variable on the left hand side. We start with a rule for a non-recursive method and later on we show how to deal with recursion.

Definition 5.24

For the statement $x \leftarrow e_0!m(e_1, \dots, e_n)$, where $x \in IVar_{d_0}^c$, $m \in MName_{d_0, \dots, d_n}^{c'}$, $e_0 \in Exp_c^c$, and $e_i \in Exp_{d_i}^c$ for $i = 1, \dots, n$, we have the following proof rule:

$$\frac{\{P^{c'} \wedge \bigwedge_{i=1}^k v_i \doteq \text{nil}\} \langle U | c' : S \rangle \{Q^c[e/r]\}, \quad Q[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}] \rightarrow R^c[r/x]}{\{P[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}]\} \langle U | c : x \leftarrow e_0!m(e_1, \dots, e_n) \rangle \{R\}} \quad (\text{MI})$$

where $S \in Stat^{c'}$ and $e \in Exp_{d_0}^{c'}$ are the statement and expression occurring in the definition of the method m in the unit U , u_1, \dots, u_n are its formal parameters, v_1, \dots, v_k is a row of temporary variables that are *not* formal parameters ($k \geq 0$), r is a logical variable of type d_0 that does not occur in R , \bar{f} is an arbitrary row of expressions (*not* logical expressions) in class c , and \bar{z} is a row of logical variables, mutually different and different from r , such that the type of each z_i is the same as the type of the corresponding f_i . Furthermore, $[\bar{e}/\text{self}, \bar{u}]$ stands for a *simultaneous* substitution having the “components” $[e_0/\text{self}], [e_1/u_1], \dots, [e_n/u_n]$ (a formal definition will follow). We require that no temporary variables other than the formal parameters u_1, \dots, u_n occur in P or Q .

We still have to define precisely what $[\bar{e}/\text{self}, \bar{u}]$ means, but before doing that let us give some informal explanation of the above rule. When a statement as above is executed, several things happen. First, control is transferred from the sender of the

message to the receiver (context switching). The formal parameters of the receiver are initialized with the values of the expressions that form the actual parameters of the message and the other temporary variables are initialized to nil. Then the body S of the method is executed. After that the result expression e is evaluated, control is returned to the sender, the temporary variables are restored, and the result object is assigned to the variable x .

The first thing, the context switching, is represented by the substitution $[e_0/\text{self}]$. A little more precisely, an assertion P as seen from the receiver's viewpoint is equivalent to $P[e_0/\text{self}]$ from the viewpoint of the sender. Note that this substitution also changes the class of the assertion: $P[e_0/\text{self}] \in \text{Ass}^c$ whereas $P \in \text{Ass}^{c'}$. Now the passing of the parameters is simply represented by the substitution $[e_1, \dots, e_n/u_1, \dots, u_n]$. Therefore after the parameters have been transferred to the receiver, P from the receiver's viewpoint corresponds to $P[\bar{e}/\text{self}, \bar{u}]$ as seen by the sender. (Note that we really need *simultaneous* substitution here, because u_i might occur in an e_j with $j < i$, but it should not be substituted again.) In reasoning about the body of the method we may also use the information that temporary variables that are not parameters are initialized to nil.

The second thing to note is the way the result is passed back. Here the logical variable r plays an important role. This is best understood by imagining after the body S of the method the statement $r \leftarrow e$ (which is syntactically illegal, however, because r is a *logical* variable). In the sending object one could imagine the (equally illegal) statement $x \leftarrow r$. Now if the body S terminates in a state where $Q[e/r]$ holds (a premiss of the rule) then after this "virtual" statement $r \leftarrow e$ we would have a situation in which Q holds. Otherwise stated, the assertion Q describes the situation after executing the method body, in which the result is represented by the logical variable r , everything seen from the viewpoint of the receiver. Now if we context-switch this Q to the sender's side, and if it implies $R[r/x]$, then we know that after assigning the result to the variable x (our second imaginary assignment $x \leftarrow r$), the assertion R will hold.

Now we come to the role of \bar{f} and \bar{z} . We know that during the evaluation of the method the sending object becomes blocked, that is, it cannot answer any incoming messages. Therefore its instance variables will not change in the meantime. The temporary variables will be restored after the method is executed, so these will also be unchanged and finally the symbol *self* will retain its meaning over the call. All the expressions in class c (and in particular the f_i) are built from these expressions plus some inherently constant expressions and therefore their value will not change during the call. However, the method can change the variables of other objects and new objects can be created, so that the properties of these unchanged expressions *can* change. In order to be able to make use of the fact that the expressions \bar{f} are constant during the call, the rule offers the possibility to replace them temporarily by the logical variables \bar{z} , which are automatically constant. So, in reasoning from the

receiver's viewpoint (in the rule this applies to the assertions P and Q) the value of the expression f_i is represented by z_i , and in context switching f_i comes in again by the substitution $[\bar{f}/\bar{z}]$. Note that the constancy of \bar{f} is guaranteed up to the point where the result of the method is assigned to x , and that x may occur in f_i , so that it is possible to make use of the fact that x remains unchanged right up to the assignment of the result.

Definition 5.25

Now we define formally the substitution operation $[e/self]$. First we do this for logical expressions:

$$x [e/self] = e . x$$

$$u [e/self] = u$$

$$z [e/self] = z$$

$$\text{self}[e/self] = e$$

$$l [e/self] = l \quad \text{if } l = \text{nil}, \text{true}, \text{false}, n$$

$$l . x[e/self] = (l[e/self]) . x$$

$$\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi}[e/self] = \text{if } l_0[e/self] \text{ then } l_1[e/self] \text{ else } l_2[e/self] \text{ fi}$$

$$(l_1 \doteq l_2)[e/self] = (l_1[e/self]) \doteq (l_2[e/self])$$

$$(l_1 + l_2)[e/self] = (l_1[e/self]) + (l_2[e/self])$$

$$\vdots$$

$$(l_1 < l_2)[e/self] = (l_1[e/self]) < (l_2[e/self])$$

$$\vdots$$

$$|l[e/self]| = |l[e/self]|$$

$$(l_1 \cdot l_2)[e/self] = (l_1[e/self]) \cdot l_2[e/self]$$

Now we extend this to assertions other than logical expressions:

$$(P \rightarrow Q)[e/self] = (P[e/self]) \rightarrow (Q[e/self])$$

$$(\neg P) [e/self] = \neg(P[e/self])$$

$$(\forall z P) [e/self] = \forall z (P[e/self])$$

$$(\exists z P) [e/self] = \exists z (P[e/self])$$

Lemma 5.26

Let $\sigma \in \Sigma$, $\delta \in \Delta^c$, $e \in \text{Exp}_{\mathcal{C}}^c$, and define $\beta^{c'} = \mathcal{E}[e](\delta)(\sigma)$. Let $\delta' \in \Delta^{c'}$ be such that $\delta'_{(1)} = \beta$. Then we have

1. For every logical expression $l_d^{c'}$ and every valuation ω

$$\mathcal{L}[\![l]\!](\omega)(\delta')(\sigma) = \mathcal{L}[\![l[e/self]]\!](\omega)(\delta)(\sigma).$$

2. For every assertion $P^{c'}$ and every valuation ω

$$\mathcal{A}[\![P]\!](\omega)(\delta')(\sigma) = \mathcal{A}[\![P[e/self]]\!](\omega)(\delta)(\sigma).$$

Proof

An easy induction on the complexity of l and P . □

Definition 5.27

Although the intention of simultaneous substitution is probably clear to the reader, we give its definition for the case in which we really need it here, for completeness' sake. Let $\bar{e} = e_0, \dots, e_n$ and $\bar{u} = u_1, \dots, u_n$. Then we define:

$$\begin{aligned} x \quad [\bar{e}/\text{self}, \bar{u}] &= e_0 . x \\ u_i \quad [\bar{e}/\text{self}, \bar{u}] &= e_i && \text{for } i = 1, \dots, n \\ u \quad [\bar{e}/\text{self}, \bar{u}] &= u && \text{if } u \notin \{u_1, \dots, u_n\} \\ z \quad [\bar{e}/\text{self}, \bar{u}] &= z \\ \text{self}[\bar{e}/\text{self}, \bar{u}] &= e_0 \\ l \quad [\bar{e}/\text{self}, \bar{u}] &= l && \text{if } l = \text{nil}, \text{true}, \text{false}, n \\ l . x[\bar{e}/\text{self}, \bar{u}] &= (l[\bar{e}/\text{self}, \bar{u}]) . x \end{aligned}$$

$$\text{if } l_0 \text{ then } l_1 \text{ else } l_2 \text{ fi}[\bar{e}/\text{self}, \bar{u}] = \text{if } l_0[\bar{e}/\text{self}, \bar{u}] \text{ then } l_1[\bar{e}/\text{self}, \bar{u}] \text{ else } l_2[\bar{e}/\text{self}, \bar{u}] \text{ fi}$$

$$(l_1 \doteq l_2)[\bar{e}/\text{self}, \bar{u}] = (l_1[\bar{e}/\text{self}, \bar{u}]) \doteq (l_2[\bar{e}/\text{self}, \bar{u}])$$

$$(l_1 + l_2)[\bar{e}/\text{self}, \bar{u}] = (l_1[\bar{e}/\text{self}, \bar{u}]) + (l_2[\bar{e}/\text{self}, \bar{u}])$$

$$\vdots$$

$$(l_1 < l_2)[\bar{e}/\text{self}, \bar{u}] = (l_1[\bar{e}/\text{self}, \bar{u}]) < (l_2[\bar{e}/\text{self}, \bar{u}])$$

$$\vdots$$

$$|l[\bar{e}/\text{self}, \bar{u}]| = |l[\bar{e}/\text{self}, \bar{u}]|$$

$$(l_1 \cdot l_2)[\bar{e}/\text{self}, \bar{u}] = (l_1[\bar{e}/\text{self}, \bar{u}]) \cdot l_2[\bar{e}/\text{self}, \bar{u}]$$

Now we extend this to assertions other than logical expressions:

$$(P \rightarrow Q)[\bar{e}/\text{self}, \bar{u}] = (P[\bar{e}/\text{self}, \bar{u}]) \rightarrow (Q[\bar{e}/\text{self}, \bar{u}])$$

$$(\neg P) \quad [\bar{e}/\text{self}, \bar{u}] = \neg(P[\bar{e}/\text{self}, \bar{u}])$$

$$(\forall z P) \quad [\bar{e}/\text{self}, \bar{u}] = \forall z(P[\bar{e}/\text{self}, \bar{u}])$$

$$(\exists z P) \quad [\bar{e}/\text{self}, \bar{u}] = \exists z(P[\bar{e}/\text{self}, \bar{u}])$$



Figure 2: The situation before and after sending the message (example 5.29)

Of course we also have a corresponding lemma:

Lemma 5.28

Let $\sigma \in \Sigma$, $\delta \in \Delta^c$ and $e_i \in Exp_{d_i}^c$ for $i = 0, \dots, n$ (with $d_0 \in C$). Define $\beta_i = \mathcal{E}[e_i](\delta)(\sigma)$. Let $\delta' \in \Delta^{d_0}$ be such that $\delta'_{(1)} = \beta_0$ and let $\sigma' = \sigma\{\beta_i/u_i\}_{i=1}^k$. Then we have

1. For every logical expression l^{d_0} and every valuation ω

$$\mathcal{L}[l](\omega)(\delta')(\sigma') = \mathcal{L}[l[\bar{e}/\text{self}, \bar{u}]](\omega)(\delta)(\sigma).$$

2. For every assertion P^{d_0} and every valuation ω

$$\mathcal{A}[P](\omega)(\delta')(\sigma') = \mathcal{A}[P[\bar{e}/\text{self}, \bar{u}]](\omega)(\delta)(\sigma).$$

Proof

Again a quite simple induction on the complexity of l and P . □

Example 5.29

Let us illustrate the use of the rule (MI) by a small example. Consider the unit $U = c : \langle m \Leftarrow (u_0) : x_1 \leftarrow u_0 \uparrow x_2 \rangle$ and the program $\rho = \langle U | c : x_1 \leftarrow u_1 ! m(x_2) \rangle$. We want to show

$$\{u_1 . x_1 \doteq x_1 \wedge \neg u_1 \doteq \text{self}\} \rho \{u_1 . x_1 \doteq x_2 \wedge x_1 \doteq u_1 . x_2\}.$$

So let us apply the rule (MI) with the following choices:

$$\begin{aligned}
P &= x_1 \dot{=} z_1 \wedge \neg \text{self} \dot{=} z_2 \\
Q &= x_1 \dot{=} u_0 \wedge r \dot{=} x_2 \wedge \neg \text{self} \dot{=} z_2 \\
R &= u_1 . x_1 \dot{=} x_2 \wedge x_1 \dot{=} u_1 . x_2 \\
k &= 0 \quad (\text{we shall use no } v_i) \\
f_1 &= x_1 \quad (\text{represented by } z_1 \text{ in } P \text{ and } Q) \\
f_2 &= \text{self} \quad (\text{represented by } z_2 \text{ in } P \text{ and } Q)
\end{aligned}$$

First notice that $P[u_1, x_2/\text{self}, u_0][x_1, \text{self}/z_1, z_2] = u_1 . x_1 \dot{=} x_1 \wedge \neg u_1 \dot{=} \text{self}$ so that the result of the rule is precisely what we want.

For the first premiss we have to prove

$$\{x_1 \dot{=} z_1 \wedge \neg \text{self} \dot{=} z_2\} \langle U | c : x_1 \leftarrow u_0 \rangle \{x_1 \dot{=} u_0 \wedge x_2 \dot{=} x_2 \wedge \neg \text{self} \dot{=} z_2\}.$$

This is easily done with the axiom (SAI) and the rule of consequence (which will be introduced in definition 5.39).

With respect to the second premiss, we have

$$\begin{aligned}
Q[u_1, x_2/\text{self}, u_0][x_1, \text{self}/z_1, z_2] &= u_1 . x_1 \dot{=} x_2 \wedge r \dot{=} u_1 . x_2 \wedge \neg u_1 \dot{=} \text{self} \\
R[r/x_1] &= \text{if } u_1 \dot{=} \text{self} \text{ then } r \text{ else } u_1 . x_1 \text{ fi} \dot{=} x_2 \wedge r \dot{=} u_1 . x_2
\end{aligned}$$

It is quite clear that the first implies the second, and we can use this implication as an axiom (see definition 5.38).

Lemma 5.30

The proof rule (MI) is valid.

Proof

Consider the rule as listed in definition 5.24. Let $\gamma \in \Gamma$ and suppose that the premisses are valid with respect to γ . We shall prove that the conclusion is valid with respect to γ . So let $\sigma \in \Sigma$, $\delta \in \Delta^c$, and $\omega \in \Omega$ be such that $\sigma, \delta, \omega \models P[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}]$. Let $\gamma' = \mathcal{U}[\mathcal{U}](\gamma)$ and let $\sigma' = \mathcal{P}[\langle U | c : x \leftarrow e_0!m(e_1, \dots, e_n) \rangle](\gamma)(\delta)(\sigma)$. So $\sigma' = \mathcal{S}[x \leftarrow e_0!m(e_1, \dots, e_n)](\gamma')(\delta)(\sigma)$. We have to prove $\sigma', \delta, \omega \models R$.

Let $\omega' = \omega\{\mathcal{E}[f_i](\delta)(\sigma)/z_i\}_{i=1}^{|\bar{f}|}$. Then lemma 5.12 gives us $\sigma, \delta, \omega' \models P[\bar{e}/\text{self}, \bar{u}]$. Let $\beta_i = \mathcal{E}[e_i](\delta)(\sigma)$ for $i = 0, \dots, n$ and suppose that $\beta_0 \neq \perp$ and $\beta_0 \notin \delta_{(2)(c)}$ (otherwise we would have that $\sigma' = \perp$ and the result would be trivial). Define $\delta' = \langle \beta_0, \delta_{(2)}\{\delta_{(2)(c)} \cup \{\delta_{(1)}\}/c\} \rangle$ and $\sigma_1 = \langle \sigma_{(1)}, \sigma_{(2)}, \sigma_{(3)} \rangle$ where $\sigma_{(3)(d_i)}(u_i) = \beta_i$ and $\sigma_{(3)(d)}(u_d) = \perp$ if $u \notin \{u_1, \dots, u_n\}$. Now because of lemma 5.28 and the fact that temporary variables other than the u_i may not occur in P , we have $\sigma_1, \delta', \omega' \models P$. We also know that $\sigma_1, \delta', \omega' \models v_i \dot{=} \text{nil}$ for $i = 1, \dots, k$ so $\sigma_1, \delta', \omega' \models P \wedge \bigwedge_{i=1}^k v_i \dot{=} \text{nil}$.

$$\begin{array}{c}
\sigma, \delta, \omega \models P[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}] \Rightarrow \sigma, \delta, \omega' \models P[\bar{e}/\text{self}, \bar{u}] \Rightarrow \sigma_1, \delta', \omega' \models P \\
\Downarrow \\
\sigma_2, \delta', \omega' \models Q[e/r] \\
\Downarrow \\
\sigma'', \delta, \omega_1 \models Q[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}] \Leftarrow \sigma'', \delta, \omega'_1 \models Q[\bar{e}/\text{self}, \bar{u}] \Leftarrow \sigma_2, \delta', \omega'_1 \models Q \\
\Downarrow \\
\sigma'', \delta, \omega_1 \models R[r/x] \\
\Downarrow \\
\sigma', \delta, \omega_1 \models R \\
\Downarrow \\
\sigma', \delta, \omega \models R
\end{array}$$

Figure 3: The structure of the proof of lemma 5.30.

Now because of the construction of γ' in definition 3.17 we know that $\gamma'(c', \bar{d})(m) = \mathcal{M}[(\bar{u}) : S \uparrow e](\gamma')$ so we can refer directly to the method definition of m in U to see what $\gamma'(c', \bar{d})(m)$ does. So let us take $\sigma_2 = \mathcal{P}[(U|c' : S)](\gamma)(\delta')(\sigma_1)$, then $\sigma_2 = S[S](\gamma')(\delta')(\sigma_1)$. Assume that $\sigma_2 \neq \perp$, otherwise we have $\sigma' = \perp$ and we are ready. The validity of the first premiss with respect to γ tells us that $\sigma_2, \delta', \omega' \models Q[e/r]$. Let $\beta = \mathcal{E}[e](\delta')(\sigma_2)$, $\omega_1 = \omega\{\beta/r\}$, and $\omega'_1 = \omega'\{\beta/r\}$. Then because of lemma 5.12 we have $\sigma_2, \delta', \omega'_1 \models Q$.

Let $\sigma'' = \langle \sigma_{2(1)}, \sigma_{2(2)}, \sigma_{2(3)} \rangle$ (we restore the temporary variables). Now we appeal to the reader's understanding of the semantics of the language to see that the method destination e_0 , the actual parameters e_1, \dots, e_n and the expressions \bar{f} are unchanged in σ'' in comparison with σ . Otherwise stated, $\mathcal{E}[e_i](\delta)(\sigma) = \mathcal{E}[e_i](\delta)(\sigma'')$ and the same for f_i . (Of course, this can also be proved formally.) Then we know from lemma 5.28 that $\sigma'', \delta, \omega'_1 \models Q[\bar{e}/\text{self}, \bar{u}]$ and from lemma 5.12 together with the observation that $\omega'_1 = \omega_1\{\mathcal{E}[f_i](\delta)(\sigma)/z_i\}_{i=1}^{|\bar{f}|}$ we get $\sigma'', \delta, \omega_1 \models Q[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}]$.

From the second premiss we can conclude that $\sigma'', \delta, \omega_1 \models R[r/x]$. Now for the final state σ' we know that $\sigma' = \sigma''\{\beta/\delta_{(1)}, x\}$, so lemma 5.8 tells us that $\sigma', \delta, \omega_1 \models R$. Finally, because r does not occur in R , we have $\sigma', \delta, \omega \models R$. \square

Definition 5.31

For the statement $u \leftarrow e_0!m(e_1, \dots, e_n)$, where $u \in TVar_{d_0}$, $m \in MName_{d_0, \dots, d_n}^{c'}$, $e_0 \in Exp_{c'}$ and $e_i \in Exp_{d_i}^c$ for $i = 1, \dots, n$, we have the proof rule (MT) which is identical to the rule (MI) introduced in definition 5.24, except that the instance variable x is replaced everywhere by the temporary variable u .

Lemma 5.32

The proof rule (MT) is valid.

Proof

This can be proved by a slight adaptation of the proof of lemma 5.30. □

Now we come to the issues of how to handle recursive and even mutually recursive methods. For this we use an adapted version of the classical recursion rule (see for example [3]). The classical rule goes as follows (in the notation of [3]):

$$\frac{\{p\}P\{q\} \vdash \{p\}S_0\{q\}}{\{p\}P\{q\}}$$

The idea is to prove (the operator \vdash expresses provability) the correctness of the body (S_0) from the assumption that the procedure call (P) itself satisfies its specification. If that has been done we can conclude the correctness of the procedure call without assumptions. The validity of this rule can be proved as follows: the meaning of the procedure call is the limit of a increasing sequence starting with \perp , in which every element is obtained from the previous one by assuming the previous as the meaning of the procedure call and calculating the meaning of the body from that. From the premiss of the rule we can prove that every element in the sequence satisfies the specification and by a continuity argument we conclude that the procedure call itself satisfies the specification.

There are several remarks to be made. One is that in proving the premiss of the rule we may not make use of the declaration of P , because otherwise we are not sure that the implication also holds for the intermediate elements in the approximating sequence. The second remark is that if we have a non-recursive rule like our rules (MI) and (MT), then we could change the conclusion of the recursion rule into $\{p\}S\{q\}$, from which we could infer $\{p\}P\{q\}$ by the non-recursive rule. We do that in our proof system to be able to use the outcome of the recursion rule for different values of the parameters. Finally it is clear how to extend the rule to several, mutually recursive procedures.

Definition 5.33

For mutually recursive methods m_1, \dots, m_n we have the following rule:

$$\frac{\tilde{F}_1, \dots, \tilde{F}_n \quad F_1, \dots, F_n \vdash F'_1, \dots, F'_n}{F} \quad (\text{MR})$$

where

$$\begin{aligned}
F_i &= \{P_i^{c_i}[\bar{e}^i/\text{self}, \bar{u}^i][\bar{f}^i/\bar{z}^i]\} \langle U^- | c_i : x_i \leftarrow e_0^i!m_i(e_1^i, \dots, e_{n_i}^i) \rangle \{R_i^{c_i}\} \\
F'_i &= \{P_i^{c_i} \wedge \bigwedge_{j=1}^{k_i} v_j^i \doteq \text{nil}\} \langle U^- | c'_i : S_i \rangle \{Q_i^{c'_i}[e_i/r_i]\} \\
\tilde{F}_i &= Q_i[\bar{e}^i/\text{self}, \bar{u}^i][\bar{f}^i/\bar{z}^i] \rightarrow R_i[r_i/x_i] \\
F &= \{P_1 \wedge \bigwedge_{j=1}^{k_1} v_j^1 \doteq \text{nil}\} \langle U | c'_1 : S_1 \rangle \{Q_1^{c'_1}[e_1/r_1]\} \\
\bar{u}^i, S_i, \text{ and } e_i &\text{ are as they occur in the definition of } m_i \text{ in } U \\
x_i &\text{ are instance or temporary variables} \\
U^- &\text{ results from } U \text{ by deleting the definitions of } m_1, \dots, m_n \\
P_i, Q_i, R_i, \bar{f}^i, \bar{z}^i, \bar{v}^i, k_i, \text{ and } r_i &\text{ are just like in definition 5.24}
\end{aligned}$$

We cannot prove the validity of this proof rule on its own, because it depends on what the other rules can prove (the operator \vdash occurs in the premiss).

5.4 Other axioms and rules

Finally in this subsection we shall list the remaining axioms and rules of our proof system. They will deal with the more ordinary statements and therefore they are not very new (most of them can already be found in [4]).

Definition 5.34

For a side effect expression s_d^c functioning as a statement we have the following rule:

$$\frac{\{P^c\} \langle U | c : u_d \leftarrow s_d^c \rangle \{Q^c\}}{\{P\} \langle U | c : s \rangle \{Q\}} \quad (\text{ES})$$

where u_d is a temporary variable not occurring in P or Q .

Definition 5.35

For the sequential composition of statements we have the following proof rule:

$$\frac{\{P^c\} \langle U | S_1^c \rangle \{Q^c\} \quad \{Q^c\} \langle U | S_2^c \rangle \{R^c\}}{\{P\} \langle U | S_1; S_2 \rangle \{R\}} \quad (\text{SC})$$

Definition 5.36

For the conditional statement we have this rule:

$$\frac{\{P^c \wedge e_{\text{Bool}}^c\} \langle U | S_1^c \rangle \{Q^c\} \quad \{P^c \wedge \neg e\} \langle U | S_2^c \rangle \{Q^c\}}{\{P\} \langle U | \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi} \rangle \{Q\}} \quad (\text{C})$$

Definition 5.37

For the while loop we have the following rule:

$$\frac{\{P^c \wedge e_{\text{Bool}}^c\} \langle U|S^c \rangle \{P^c\}}{\{P\} \langle U|\text{while } e \text{ do } S \text{ od} \rangle \{P \wedge \neg e\}} \quad (\text{W})$$

Definition 5.38

For every *valid* (see definition 4.15) assertion P^c we have the axiom:

$$P \quad (\text{TR})$$

Definition 5.39

Finally, we have the so-called rule of consequence:

$$\frac{P_1^c \rightarrow P_2^c \quad \{P_2\} \langle U|c : S \rangle \{Q_2\} \quad Q_2^c \rightarrow Q_1^c}{\{P_1\} \langle U|c : S \rangle \{Q_1\}} \quad (\text{RC})$$

Theorem 5.40

The proof system consisting of the axioms (SAT), (SAI), (NT), (NI), and (TR), plus the rules (MI), (MT), (MR), (ES), (SC), (C), (W), and (RC) is sound, that is, for every row of correctness formulae F_0, \dots, F_n and for every environment γ we have if $F_1, \dots, F_n \vdash F_0$ and $\gamma \models F_i$ for $i = 1, \dots, n$ then $\gamma \models F_0$.

Proof

For all rules except (MR) the validity can be proved individually. For some we have already done that, for the others it is very easy. The rest of the proof runs by induction on the length of the proof of F_0 from F_1, \dots, F_n . The only interesting case occurs if the last rule applied is (MR). From now on let us use the notation of definition 5.33 and forget about the old F_0, \dots, F_n .

In the premiss of the rule (MR) we first have $\tilde{F}_1, \dots, \tilde{F}_n$, and these are valid because the only way to get them is by using the axiom (TR). The second premiss says that $F_1, \dots, F_n \vdash F'_1, \dots, F'_n$. This must be provable by a shorter proof than our current one so the induction hypothesis says that for every environment γ such that $\gamma \models F_1, \dots, F_n$ we also have that $\gamma \models F'_1, \dots, F'_n$. Let us take a particular γ and define $\gamma' = \mathcal{U}[\![U]\!](\gamma)$. Now γ' is the limit of an increasing sequence $\gamma'_0, \gamma'_1, \dots$ where $\gamma'_0 = \gamma' \{ \lambda \tilde{\beta}. \lambda \delta. \lambda \sigma. \langle \perp, \perp \rangle / m_i \}_{i=1}^n$ and γ'_{i+1} is obtained from γ'_i by calculating and filling

in the meanings of the method definitions of m_1, \dots, m_n . Furthermore we observe that for every i and for every $m \in \{m_1, \dots, m_n\}$ we have that $\mathcal{U}[U^-](\gamma'_i)(m) = \gamma'_i(m)$ because m is not defined in U^- .

Now for γ'_0 we have quite trivially that $\gamma'_0 \models F'_1, \dots, F'_n$ (the send-expression never terminates). Furthermore from $\gamma'_i \models F'_j$ we can get to $\gamma'_{i+1} \models F_j$ by an argument analogous to that in lemma 5.30. From the validity of the second premiss we can then conclude that $\gamma'_{i+1} \models F'_j$ for $j = 1, \dots, n$. By induction we get $\gamma'_i \models F'_1, \dots, F'_n$ for every i , so by continuity we get in particular $\gamma' \models F'_1$. And this in turn implies $\gamma \models F$. \square

6 Completeness

6.1 Introduction

We prove in this section that every valid correctness formula about an arbitrary closed program is derivable from the proof system based on the assertion language with quantification over finite sequences of objects. To this end we use enhanced versions of the standard techniques for proving completeness. These techniques are based on the expressibility of the *strongest postcondition*, or, alternatively, the *weakest precondition*. Using the assertion language with quantification over finite sequences of objects we know how to express the strongest postcondition. However, we conjecture that we cannot in general express the strongest postcondition or the weakest precondition within the assertion language with recursive predicates. We think this is due to the inexpressibility within this assertion language of the notion of *finiteness*.

In order to get a complete proof system, however, we have to modify the rules (MI), (MT), and (MR) so that we can reason about *deadlock behaviour*. Regardless of the assertion language we use these rules are incomplete. Consider the following example:

Example 6.1

Let $\rho = \langle U | c : x \leftarrow \text{self}!m() \rangle$ be closed and $m() \Leftarrow \text{nil} \uparrow \text{nil}$ occur in U . We obviously have $\models \{\text{true}\} \rho \{\text{false}\}$. But we do *not* have the derivability of this correctness formula. For otherwise there would exist assertions P, Q and R such that:

1. $\vdash \{P \wedge \bigwedge_i v_i \doteq \text{nil}\} \langle U | c : \text{nil} \rangle \{Q[\text{nil}/\tau]\}$
2. $\models Q[\text{self}/\text{self}][\bar{f}/\bar{z}] \rightarrow R[r/x]$
3. $\models \text{true} \rightarrow P[\text{self}/\text{self}][\bar{f}/\bar{z}]$ and $\models R \rightarrow \text{false}$

for some sequence of expressions \bar{f} , sequence of corresponding logical variables \bar{z} and logical variable r of the same type as the instance variable x . Now, as $\models R \rightarrow \text{false}$, we have $\models R[r/x] \rightarrow \text{false}$. So from clause 2 it then follows that $\models Q[\text{self/self}][\bar{f}/\bar{z}] \rightarrow \text{false}$. Furthermore we have $\models Q[\text{self/self}] \leftrightarrow Q$ so we infer $\models Q[\bar{f}/\bar{z}] \rightarrow \text{false}$. From clause 1 in turn it is not difficult to deduce that $\models P \rightarrow Q[\text{nil}/r]$ (use $\bar{v}_i \cap TVar(P, Q) = \emptyset$ and the truth of the correctness formula of clause 1). So we have $\models P[\bar{f}/\bar{z}] \rightarrow Q[\text{nil}/r][\bar{f}/\bar{z}]$. Note next that $\models Q[\bar{f}/\bar{z}] \rightarrow \text{false}$ implies $\models Q[\text{nil}/r][\bar{f}/\bar{z}] \rightarrow \text{false}$, from which we infer that $\models P[\bar{f}/\bar{z}] \rightarrow \text{false}$, which in turn, using $\models P[\text{self/self}] \leftrightarrow P$, would imply by clause 3 $\models \text{true} \rightarrow \text{false}$. We thus have reached a contradiction. So we conclude that $\not\models \{ \text{true} \} \rho \{ \text{false} \}$.

Note that adding the conjunct $\neg(\text{self} \doteq e_0)$ to the precondition of the conclusion of the rules (MI) and (MT) does not solve the general case of longer cycles in the calling chain.

To reason about deadlock in the proof system based on the assertion language containing quantification over finite sequences we introduce a collection of logical variables with special roles.

Definition 6.2

We fix for each class name c a logical variable $b_c \in LVar_c$. Furthermore we define $BVar = \{b_c : c \in C\}$.

We will interpret the variable b_c as denoting a sequence of all the blocked objects of class c . Formally, we redefine the notion $OK(\sigma, \delta, \omega)$ as follows:

Definition 6.3

For arbitrary σ, δ, ω we define $OK(\sigma, \delta, \omega)$ iff σ is consistent, δ agrees with σ , ω is compatible with σ and for an arbitrary c we have

$$\delta_{(2)(c)} = \{ \alpha : \exists n \in \mathbb{N} (elt(b_c, n) = \alpha \neq \perp) \}.$$

So we have $OK(\sigma, \delta, \omega)$ if additionally b_c , for an arbitrary c , consists precisely of all the blocked objects of class c . Note that we have thus introduced in the assertion language a means to refer to the second component of a context. Given this fixed interpretation we do not allow the variable b_c to be quantified. It is a straightforward exercise to check that under this definition of $OK(\sigma, \delta, \omega)$ the soundness proofs given still hold.

Next we modify the rule (MI) as follows:

Definition 6.4

For the statement $x \leftarrow e_0!m(e_1, \dots, e_n)$, where $x \in IVar_{d_0}^c$, $m \in MName_{d_0, \dots, d_n}^{c'}$,

$e_0 \in \text{Exp}_c^c$, and $e_i \in \text{Exp}_{d_i}^c$ for $i = 1, \dots, n$, we have the following proof rule:

$$\frac{\{P^{c'} \wedge \bigwedge_{i=1}^k v_i \doteq \text{nil} \wedge \neg(\text{self} \in b_{c'})\} \langle U|c' : S \rangle \{Q^{c'}[e/r]\}, \quad Q' \rightarrow R^c[r/x]}{\{P'\} \langle U|c : x \leftarrow e_0!m(e_1, \dots, e_n) \rangle \{R\}} \quad (\text{MI})$$

where $P' = P[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}][b_c \circ \langle \text{self} \rangle / b_c]$, $Q' = Q[\bar{e}/\text{self}, \bar{u}][\bar{f}/\bar{z}][b_c \circ \langle \text{self} \rangle / b_c]$, $S \in \text{Stat}^{c'}$ and $e \in \text{Exp}_{d_0}^{c'}$ are the statement and expression occurring in the definition of the method m in the unit U , u_1, \dots, u_n are its formal parameters, v_1, \dots, v_k is a row of temporary variables that are *not* formal parameters ($k \geq 0$), r is a logical variable of type d_0 that does not occur in R , \bar{f} is an arbitrary row of expressions (*not* logical expressions) in class c , and \bar{z} is a row of logical variables, mutually different and different from r , such that the type of each z_i is the same as the type of the corresponding f_i . We require that no temporary variables other than the formal parameters u_1, \dots, u_n occur in P or Q . The boolean expression $l_1 \in l_2$ abbreviates $\exists i(l_1 \doteq l_2 \cdot i)$, where i is some fresh logical integer variable. $P[b_c \circ \langle \text{self} \rangle / b_c]$, for an arbitrary assertion P , equals the assertion

$$\exists z(P[z/b_c] \wedge |z| = |b_c| + 1 \wedge \forall i(i \leq |b_c| \rightarrow z \cdot i \doteq b_c \cdot i) \wedge (z \cdot |z| \doteq \text{self}))$$

where $z \in \text{LVar}_{c^*}$, $i \in \text{LVar}_{\text{Int}}$ are some fresh variables.

The idea of this substitution $[b_c \circ \langle \text{self} \rangle / b_c]$ can be explained roughly as follows: Occurrences of the variable b_c in the assertions $P^{c'}$ and $Q^{c'}$, which describe the input state and the output state of the receiver of the method call, denote the set of blocked objects of class c belonging to those states. When we want to describe the input state and the output state of the receiver from the point of view of the sender we have to take into account that this set of blocked objects can now be viewed as the set of blocked objects of class c belonging to the input state and the output state of the sender of the method call plus the sender itself.

The rules (MT) and (MR) are modified accordingly. The soundness proofs of these new versions of (MI) and (MT) are straightforward modifications of the proofs of the soundness of the original ones (in the proof of 5.30 the substitution $[b_c \circ \langle \text{self} \rangle / b_c]$ can be considered simply as part of the simultaneous substitution $[\bar{f}/\bar{z}]$). The proof of the soundness of the new version of (MR), assuming the soundness of the new versions of (MI) and (MT), does not need to be modified.

We note that with respect to the proof system based on the assertion language containing recursive predicates this proof method does not apply. To incorporate some reasoning mechanism about deadlock behaviour in this system one could add to it some notion of auxiliary variables, which can be used to code the relevant control information.

It will appear to be technically convenient to introduce another modification of the rule (MR). This modification consists simply of replacing every occurrence of U^- in

this rule by U itself. We denote the resulting rule by (NMR). The main difference between the rules (NMR) and (MR) is that the rule (NMR) allows nested applications to some method name. However, in appendix A it is shown that a proof using the rule (NMR) can be transformed into a proof using (MR), and vice versa.

To be able to prove completeness we have to add the following rules to the proof system (based on the assertion language containing quantification over finite sequences).

Definition 6.5

Conjunction rule:

$$\frac{\{P_1^c\}\rho^c\{Q_1^c\} \quad \{P_2^c\}\rho^c\{Q_2^c\}}{\{P_1^c \wedge P_2^c\}\rho^c\{Q_1^c \wedge Q_2^c\}} \quad (\text{CR})$$

Definition 6.6

Elimination rule 1:

$$\frac{\{P^c\}\rho^c\{Q^c\}}{\{\exists z_d P^c \vee P[\text{nil}/z_d]\}\rho^c\{Q^c\}} \quad (\text{ER1})$$

where $z_d \notin LVar(Q^c) \cup BVar$. Due to the interpretation of the quantifiers as ranging only over existing objects we have to express explicitly that the precondition also holds when the value of the quantified variable is undefined (nil).

Definition 6.7

Elimination rule 2:

$$\frac{\{P^c\}\rho^c\{Q^c\}}{\{\exists z_a P^c\}\rho^c\{Q^c\}} \quad (\text{ER2})$$

where $a \vdash d^*$, for some d , and $z_a \notin LVar(Q^c) \cup BVar$.

Definition 6.8

Initialization rule 1:

$$\frac{\{P^c\}\rho^c\{Q^c\}}{\{P^c[l/z]\}\rho^c\{Q^c\}} \quad (\text{IR1})$$

where z and l are of the same type, and $z \notin LVar(Q^c) \cup BVar$.

Definition 6.9

Initialization rule 2:

$$\frac{\{P^c\}\rho^c\{Q^c\}}{\{P^c[l/u]\}\rho^c\{Q^c\}} \quad (\text{IR2})$$

where u and l are of the same type and $u \notin TVar(\rho, Q)$.

Definition 6.10

Substitution rule:

$$\frac{\{P^c\}\rho^c\{Q^c\}}{\{P^c[z'/z]\}\rho^c\{Q^c[z'/z]\}} \quad (\text{SR})$$

where z', z are logical variables of the same type, and $z \notin BVar$.

The soundness of these new rules is a straightforward exercise. We illustrate the necessity of the condition $z \notin BVar$ by the following example:

Example 6.11

Let $\rho = \langle U|c' : y \leftarrow x!m() \rangle$. By the new definition of $OK(\sigma, \delta, \omega)$ we have, assuming the type of the variable x to be c ,

$$\models \{x \in b_c\}\rho\{\text{false}\},$$

where $x \in b_c$ abbreviates the assertion $\exists i(x \doteq b_c \cdot i)$. If we would allow the initialization of the variable b_c , or allow it to be substituted, we could derive from this formula by an application of the rule (SR) or (IR1) the following:

$$\{x \in z\}\rho\{\text{false}\}.$$

Applying next the elimination rule (ER2), assuming $z \notin BVar$, then gives us the derivability of the formula:

$$\{\exists z(x \in z)\}\rho\{\text{false}\}.$$

Finally, we apply the consequence rule:

$$\{\text{true}\}\rho\{\text{false}\}.$$

But this last formula is not valid in general!

Finally, for technical convenience we would like to assume that the sets C , $IVar$, and $TVar$ are finite. This assumption can be justified as follows: Let C' be a finite subset of C , and $IVar'$ be a finite subset of $\bigcup_{c,d} IVar_d^c$, where c ranges over C' , and d ranges over the set $C'^+ = C' \cup \{\text{Int}, \text{Bool}\}$. Next we fix the temporary integer variables u, u' , and for every $d \in C'^+$ the temporary variables re_d, re'_d . Let \bar{re} denote a sequence of these variables. Now let $TVar'$ be a finite subset of $\bigcup_d TVar_d$ (again, d ranging over C'^+), such that $\bar{re} \subseteq TVar'$. Given these sets C' , $IVar'$, and $TVar'$ we have the following definition.

Definition 6.12

We define an expression l_a^c to be *restricted* iff $c \in C'$, $a = d, d^*$, with $d \in C'^+$, $IVar(l_a^c) \subseteq IVar'$, and $TVar(l_a^c) \subseteq TVar'$. We define an assertion P^c to be restricted iff $c \in C'$ and every expression occurring in P^c is restricted. We call a program

$\rho = \langle U|c : S \rangle$ restricted iff $c \in C'$, every expression occurring in ρ is restricted, $u, u' \notin TVar(\rho)$, and, finally, the temporary variables re_d, re'_d are only allowed in the main statement S itself, where $S = re_d \leftarrow s_d$ or $S = re'_d \leftarrow s_d$, with $TVar(s) \cap \bar{re} = \emptyset$. A correctness formula $\{P\}\rho\{Q\}$ is called restricted iff P, Q , and ρ are restricted.

We will prove that an arbitrary valid restricted correctness formula is derivable by a derivation in which there occur only restricted correctness formulae. Such a derivation we call restricted too. The extra variables \bar{re} are used in applications of the rules (W) and (ES): The variables re_d, re'_d are used to store temporarily the result of the execution of a statement s_d ; the variables u, u' are needed to express the *invariant* of a while statement. However applications of the consequence rule in a restricted derivation are based on a different notion of validity of assertions and correctness formulae. This new notion of validity consists of restricting all the semantic entities to the sets $C', IVar'$, and $TVar'$. As an example of the restriction of a semantic entity we define that of a state.

Definition 6.13

We define the restriction of a state σ , which we denote by $\sigma \downarrow$, to be an element of

$$\Sigma \downarrow = \prod_{c \in C'} \mathbf{P}^c \times \prod_{c \in C', d \in C'^+} (\mathbf{O}^c \rightarrow IVar_d^c \rightarrow \mathbf{O}_\perp^d) \times \prod_{d \in C'^+} (TVar_d' \rightarrow \mathbf{O}_\perp^d)$$

such that

- $\sigma \downarrow^{(c)} = \sigma^{(c)}, c \in C'$.
- $\sigma \downarrow(\alpha)(x) = \sigma(\alpha)(x), \alpha \in \mathbf{O}^c, \text{ for } c \in C', \text{ and } x \in IVar'$.
- $\sigma \downarrow(u) = \sigma(u), u \in TVar'$.

In a similar way we have corresponding restricted versions of all our semantic entities. We have the following lemma, which states that the meaning of a restricted program depends only on those parts of a state specified by the sets $C', IVar'$, and $TVar'$.

Lemma 6.14

For an arbitrary restricted program ρ , and $\sigma, \sigma', \delta, \gamma$ such that

1. $\sigma^{(c)} = \sigma'^{(c)}, c \notin C'$.
2. $\sigma(\alpha) = \sigma'(\alpha), \text{ for } \alpha \in \mathbf{O}^c, c \notin C'$.
3. $\sigma(\alpha) = \sigma'(\alpha), \text{ for } \alpha \in \mathbf{O}^c \setminus \sigma'^{(c)}, c \in C'$.
4. $\sigma(\alpha)(x) = \sigma'(\alpha)(x), \text{ for } \alpha \in \sigma^{(c)}, c \in C', x \notin IVar'$.

5. $\sigma'(\alpha)(x) = \perp$, for $\alpha \in \sigma'^{(c)} \setminus \sigma^{(c)}$, $c \in C'$, $x \notin IVar'$.
6. $\sigma(u) = \sigma'(u)$, $u \notin TVar'$.

we have

$$\sigma' = \mathcal{P}[\rho](\gamma)(\delta)(\sigma) \text{ iff } \sigma' \downarrow = \mathcal{P}'[\rho](\gamma \downarrow)(\delta \downarrow)(\sigma \downarrow),$$

where \mathcal{P}' , $\gamma \downarrow$, and $\delta \downarrow$ denote the restricted versions of \mathcal{P} , γ , and δ , respectively. (Here $\sigma(\alpha)$ denotes the local state of α and $\sigma(\alpha)(x)$, x an instance variable, denotes the value of the variable x of the object α , finally, $\sigma(u)$, u a temporary variable, denotes the value of u in state σ .)

The first condition above states that σ and σ' agree with respect to the existing objects of class c , $c \notin C'$. The second condition states that σ and σ' agree with respect to the local states of objects belonging to a class c , $c \notin C'$. That the states σ and σ' agree with respect to the local states of objects belonging to a class c , $c \in C'$, which do not exist in σ' , is expressed by the third clause. The fourth clause states that σ and σ' agree with respect to the variables not belonging to $IVar'$ of objects of a class c , $c \in C'$, which exist in σ . The fifth clause then states that the value of a variable not belonging to $IVar'$ of an object of a class c , $c \in C'$, which exist in σ' but does not exist in σ , is undefined in the state σ' . The last clause states that σ and σ' agree with respect to the temporary variables not belonging to $TVar'$. These conditions are necessary to prove that if $\sigma' \downarrow = \mathcal{P}'[\rho](\gamma \downarrow)(\delta \downarrow)(\sigma \downarrow)$ then $\sigma' = \mathcal{P}[\rho](\gamma)(\delta)(\sigma)$.

Proof

Induction on the structure of the program ρ . □

By the following two lemmas we have that applications of the consequence rule occurring in a restricted derivation also apply with respect to the original notion of validity, thus justifying our assumption of the finiteness of the sets C , $IVar$, and $TVar$. These lemmas state that the truth of a restricted assertion and that of a correctness formula only depend on those parts of a state specified by the sets C' , $IVar'$, and $TVar'$.

Lemma 6.15

For an arbitrary restricted assertion P^c , and σ, δ, ω such that $OK(\sigma, \delta, \omega)$ we have

$$\sigma, \delta, \omega \models P^c \text{ iff } \sigma \downarrow, \delta \downarrow, \omega \downarrow \models P^c,$$

where $\omega \downarrow \in \prod_a LVar_a \rightarrow \mathbf{O}_\perp^a$, with a ranging over the set $\{d, d^* : d \in C'^+\}$, and $\omega \downarrow(z) = \omega(z)$.

Proof

Straightforward induction on the structure of P^c . □

Furthermore we have

Lemma 6.16

Let σ, δ, ω such that $OK(\sigma, \delta, \omega)$. We have for an arbitrary restricted correctness formula $\{P\}_\rho\{Q\}$

$$\sigma, \delta, \omega \models \{P\}_\rho\{Q\} \text{ iff } \sigma \downarrow, \delta \downarrow, \omega \downarrow \models \{P\}_\rho\{Q\}.$$

Proof

Straightforward, using lemmas 6.14 and 6.15. \square

So in the sequel we may assume the sets C , $IVar$, and $TVar$ to be finite. Further, we assume given a set of temporary variables \tilde{re} as defined above. A program ρ from now on will denote, when not stated otherwise, a program such that the temporary variables re, re' are allowed to occur in it only in assignments $re \leftarrow s, re' \leftarrow s$, with $re, re' \notin TVar(s)$, and $u, u' \notin TVar(\rho)$. This concludes our discussion concerning the justification of the assumption of the finiteness of the sets C , $IVar$, and $TVar$.

6.2 The strongest postcondition

To be able to prove completeness we first have to analyze the notion of a *strongest postcondition* and its expressibility in the assertion language. As noted already in the introduction, the expressibility of the strongest postcondition in the assertion language with recursive predicates is still an open problem and so is the completeness of the proof system based on this assertion language.

For the analysis of the notion of a strongest postcondition we need some definitions and a theorem. We start with the following definition:

Definition 6.17

An *object-space isomorphism* (*osi*) is a family of functions $f = \langle f^d \rangle_{d \in C^+}$, where $f^d \in \mathbf{O}_\perp^d \rightarrow \mathbf{O}_\perp^d$ is a bijection, $f^d(\perp) = \perp$ and f^d , for $d = \text{Int}, \text{Bool}$, is the identity mapping.

Given an *osi* f we next define the *isomorphic image* of an arbitrary state.

Definition 6.18

Given an *osi* f we define for an arbitrary state σ the state $f(\sigma)$ as follows:

- For every c : $f(\sigma)^{(c)} = f^c(\sigma^{(c)})$.
- For every c, d, α^c, x_d^c : $f(\sigma)(\alpha)(x_d^c) = f^d(\sigma(f^{-1c}(\alpha))(x_d^c))$, where the *osi* f^{-1} denotes the *inverse* of f : $f^{-1} = \langle (f^d)^{-1} \rangle_d$.

- For every d, u_d : $f(\sigma)(u_d) = f^d(\sigma(u_d))$.

Here $f^c(X)$, for some $X \subseteq O^c$, denotes the set $\{f^c(\alpha) : \alpha \in X\}$.

The following theorem essentially expresses that states which are isomorphic cannot be distinguished by the assertion language.

Theorem 6.19

Let f be an *osi* and σ, δ, ω be such that $OK(\sigma, \delta, \omega)$. Then for every logical expression l_a^c and assertion P^c we have:

- $f^a(\mathcal{L}[l_a^c](\omega)(\delta)(\sigma)) = \mathcal{L}[l_a^c](f(\omega))(f(\delta))(f(\sigma))$,
- $\mathcal{A}[P^c](\omega)(\delta)(\sigma) = \mathcal{A}[P^c](f(\omega))(f(\delta))(f(\sigma))$.

where $f(\delta)_{(1)} = f^c(\delta_{(1)})$,

$f(\delta)_{(2)(c')} = f^{c'}(\delta_{(2)(c')})$, for an arbitrary c' , and

$f(\omega)(z_d) = f^d(\omega(z_d))$, $(f^{d^*}(\langle \alpha_1, \dots, \alpha_n \rangle) = \langle f^d(\alpha_1), \dots, f^d(\alpha_n) \rangle)$.

Proof

Straightforward induction on the structure of l_a^c, P^c . We only treat the case $l = x_d^c$:

$$\mathcal{L}[x](f(\omega))(f(\delta))(f(\sigma)) = f(\sigma)(f^c(\delta_{(1)}))(x) = f^d(\sigma(\delta_{(1)}))(x) = f^d(\mathcal{L}[x](\omega)(\delta)(\sigma))$$

□

We are now sufficiently prepared to analyze the notion of a strongest postcondition. Given a program ρ^c and an assertion P^c , we denote by $sp(\rho^c, P^c)$ the set of final states of executions of ρ^c starting from a state satisfying P^c . An assertion, defining this set of states $sp(\rho^c, P^c)$ is called the strongest postcondition of P^c with respect to ρ^c . As established by the previous theorem, the set of states defined by an arbitrary assertion is closed under isomorphism. However, in general, given a program ρ^c and an assertion P^c , the set of states $sp(\rho^c, P^c)$ is not closed under isomorphism. Consider the following example:

Example 6.20

Take $\rho^c = \langle U|c : x \leftarrow \text{new} \rangle$, with ρ^c closed, and σ, σ', δ such that $\sigma'^{(c)} = \{\alpha, \beta\}$, $\sigma^{(c)} = \{\alpha\}$, $\delta_{(1)} = \alpha$ and $\sigma' = \mathcal{P}[\rho^c](\gamma)(\delta)(\sigma)$. Let $P^c = \text{true}$. So we have that $\text{pick}^c(\{\alpha\}) = \beta$. Let f be an arbitrary *osi* such that $\text{pick}^c(\{f^c(\alpha)\}) \neq f^c(\beta)$ and $\text{pick}^c(\{f^c(\beta)\}) \neq f^c(\alpha)$. So we have that $f(\sigma')^{(c)} = \{f^c(\alpha), f^c(\beta)\}$. Now suppose that there is a σ_0 such that $f(\sigma') = \mathcal{P}[\rho^c](\gamma)(f(\delta))(f(\sigma_0))$. Then we would have $\sigma_0^{(c)} = \{f^c(\alpha)\}$ or $\sigma_0^{(c)} = \{f^c(\beta)\}$, but both cases lead to a contradiction. Therefore such a σ_0 does not exist and $f(\sigma') \notin sp(\rho^c, \text{true})$.

This discrepancy between the assertion language and the semantics of the programming language is solved by closing this set $sp(\rho^c, P^c)$ under isomorphism. Of course it is not immediately clear that this will work! We will see later that we indeed encounter some difficulties in the completeness proof due to this. These difficulties require some additional reasoning not present in the standard completeness proofs. The following theorem states the existence of an assertion defining the closure under isomorphism of the set $sp(\rho^c, P^c)$.

Theorem 6.21

Let ρ^c be closed (not necessarily restricted), $BVar \subseteq L \subseteq LVar$ (L finite), P^c such that $LVar(P^c) \subseteq L$. Then there exists an assertion $SP_L^c(\rho, P^c)$ such that $LVar(SP_L^c(\rho, P^c)) \subseteq L$ and for σ, δ, ω such that $OK(\sigma, \delta, \omega)$ we have:

$$\sigma, \delta, \omega \models SP_L^c(\rho, P^c)$$

iff there exist an *osi* f and a state σ_0 such that:

- $f(\sigma) = \mathcal{P}[\rho](\gamma)(\delta')(\sigma_0)$, γ arbitrary,
- $\sigma_0, \delta', \omega' \models P^c$,

where $\delta' = f(\delta)$ and $\omega' = f(\omega) \downarrow L$. Here we define

$$\begin{aligned} (f(\omega) \downarrow L)(z) &= f(\omega(z)) \quad z \in L \\ &= \perp \quad z \in (LVar \cap \bigcup_d LVar_d) \setminus L \\ &= \epsilon \quad z \in (LVar \cap \bigcup_d LVar_{d^*}) \setminus L. \end{aligned}$$

Note that in the above theorem we cannot take $f(\omega)$, where $f(\omega)(z) = f(\omega(z))$, for ω' . This would require that $f(\omega)$ and σ_0 are compatible, which cannot be expressed by our assertion language. For suppose there exists an $\alpha \in \sigma^{(c')}$, for some c' , such that $f^{c'}(\alpha) \notin \sigma_0^{(c')}$. Let $z_{c'} \notin L$, it then follows that $\sigma, \delta, \omega\{\alpha/z_{c'}\} \models SP_L^c(\rho, P^c)$, but on the other hand it is not the case that $f(\omega\{\alpha/z_{c'}\})$ and σ_0 are compatible, so we do not have $\sigma_0, \delta', f(\omega\{\alpha/z_{c'}\}) \models P^c$. Note that the above argument essentially boils down to the fact that we cannot describe by one assertion the values of infinitely many logical variables. Thus we have to specify a finite set of logical variables L such that the restriction of $f(\omega)$ to this set L is compatible with σ_0 .

Proof

See appendix B. □

The following two lemmas together state the correctness of our definition of the notion of strongest postcondition.

Lemma 6.22

For an arbitrary $BVar \subseteq L \subseteq LVar$ (L finite), closed program ρ^c and assertion P^c such that $LVar(P^c) \subseteq L$, we have

$$\models \{P^c\} \rho \{SP_L^c(\rho, P^c)\}.$$

Proof

Let $\sigma, \sigma', \delta, \omega$ ($\sigma, \sigma' \neq \perp$) be such that $OK(\sigma, \delta, \omega)$, $\sigma' = \mathcal{P}^c[\rho^c](\gamma)(\delta)(\sigma)$ (γ arbitrary), and $\sigma, \delta, \omega \models P^c$. We have that $\sigma', \delta, \omega \models SP_L^c(\rho, P^c)$, for take for the *osi* f the family of identity mappings, for σ_0 the state σ , and note that because $LVar(P^c) \subseteq L$ we have $\sigma, \delta, \omega' \models P^c$, where $\omega' = \omega \downarrow L$. \square

Lemma 6.23

For an arbitrary closed program ρ^c , assertions P^c, Q^c , $BVar \subseteq L \subseteq LVar$ (L finite) such that $LVar(P^c, Q^c) \subseteq L$ we have

$$\models \{P^c\} \rho^c \{Q^c\} \text{ implies } \models SP_L^c(\rho^c, P^c) \rightarrow Q^c.$$

Proof

Assume $\models \{P\} \rho \{Q\}$ and let σ, δ, ω such that $OK(\sigma, \delta, \omega)$ and $\sigma, \delta, \omega \models SP_L^c(\rho^c, P^c)$. So there exist an *osi* f and a state σ_0 such that:

- $f(\sigma) = \mathcal{P}[\rho](\gamma)(\delta')(\sigma_0)$, γ arbitrary.
- $\sigma_0, \delta', \omega' \models P^c$.

where $\delta' = f(\delta)$ and $\omega' = f(\omega) \downarrow L$. From $\models \{P^c\} \rho^c \{Q^c\}$ we then infer that $f(\sigma), \delta', \omega' \models Q^c$. By $LVar(Q^c) \subseteq L$ we have $f(\sigma), \delta', f(\omega) \models Q^c$. So by theorem 6.19 we conclude $\sigma, \delta, \omega \models Q^c$. \square

6.3 Freezing the initial state

An essential notion of the standard technique for proving completeness consists of what is called freezing the initial state. To explain this notion, let, only in this paragraph, ρ denote a program of some simple procedural language (like the ones treated in [3] or [10]) and σ, σ' denote some simple functions assigning values to program variables. Let \bar{x} denote the set of program variables occurring in ρ , \bar{z} denote a corresponding sequence of logical variables and $\bar{x} \doteq \bar{z}$ abbreviate $\bigwedge_i (x_i \doteq z_i)$. Furthermore

let $SP(\rho, \bar{x} \doteq \bar{z})$ be an assertion describing the set of final states resulting from executions of ρ starting in a state satisfying $\bar{x} \doteq \bar{z}$. In the standard completeness proof an important consequence of the definition of the notion of strongest postcondition is that the assertion $SP(\rho, \bar{x} \doteq \bar{z})$ in the following sense describes the *graph* of ρ :

- If the execution of ρ starting from the state σ results in the state σ' then $SP(\rho, \bar{x} \doteq \bar{z})$ holds in σ' when the logical variable z_i is interpreted as $\sigma(x_i)$, the value of x_i in σ .
- If $SP(\rho, \bar{x} \doteq \bar{z})$ holds in a state σ' , assuming the logical variable z_i to be interpreted as some value d_i , then there exists an execution of ρ starting from the state $\sigma'\{d_i/x_i\}_i$ which results in σ' .

Note that the logical variables \bar{z} are used to “freeze” the initial state.

Now one of the problems in applying the standard techniques for proving completeness to our proof system consists of how to store a state in a *finite* set of logical variables. A simple assertion like $\bar{x} = \bar{z}$ does not make sense, because a variable x can be evaluated only with respect to some object. To be able to construct an assertion which expresses how a state is stored in the logical environment we introduce some special logical variables. First we fix for each class name c the logical variables $cr_c, bl_c \in LVar_c$. Every existing object belonging to class c is supposed to be a member of the sequence denoted by cr_c . For convenience, we also include *nil* in cr_c . The sequence denoted by bl_c on the other hand is supposed to contain all the blocked objects belonging to class c . Furthermore for each instance variable x_d we fix a logical variable $iv_x \in LVar_d$ and, finally, for each temporary variable u_d we fix a logical variable $tv_u \in LVar_d$. The sequence denoted by iv_x , $x \in IVar^c$, will store the value of the variable x for every existing object belonging to class c in the following way: Every existing object of class c occurs at least once in the sequence denoted by cr_c . Now the i th element of the sequence iv_x is the value of the variable x in the object that is the i th element of the sequence cr_c . The value of tv_u , $u \in TVar$, just equals that of u .

All these newly introduced logical variables we assume to be distinct. We let \bar{st} denote a particular sequence (without repetitions) of these logical variables. Now we are ready to define formally the assertion *init*, which expresses that the current state is represented by \bar{st} . In other words, *init* is our analogue of the assertion $\bar{x} = \bar{z}$.

Definition 6.24

We define the assertion *init* as follows:

$$\begin{aligned} init = & \bigwedge_c cr_c \cdot 1 \doteq nil \wedge \forall z_c \exists i (z_c \doteq cr_c \cdot i) \quad \wedge \\ & \bigwedge_c \forall i (\bigwedge_{x \in IVar^c} ((cr_c \cdot i) \cdot x \doteq iv_x \cdot i)) \quad \wedge \\ & \bigwedge_{u \in TVar} (u \doteq tv_u) \quad \wedge \\ & \bigwedge_c (b_c \doteq bl_c) \end{aligned}$$

where $IV^c = \bigcup_d IVar_d^c$, $TV = \bigcup_d TVar_d$, and the logical variable i is supposed to range over the integers. Note that in our assertion language we do *not* have equality between logical expressions of type d^* , for an arbitrary d . However, these equalities can easily be expressed in the assertion language: If l_1 and l_2 are two logical expressions ranging over sequences, then $l_1 \doteq l_2$ can be expressed as $\forall i(l_1 \cdot i \doteq l_2 \cdot i)$, where i is some logical integer variable. Furthermore we remark that for every class name c we have $init \in Ass^c$.

In the following two definitions we define a transformation of a logical expression and an assertion such that the transformed versions only refer to the logical environment. Expressions referring to the state will be translated into expressions which refer to the corresponding part of the logical environment \bar{st} used to reflect the state. The problem such a transformation poses can be best explained by the following example:

Example 6.25

Suppose we want to transform the expression consisting of the instance variable x . This expression denotes the value of x with respect to the object denoted by the expression *self*. But to look up this value in the logical environment one has to know where the object denoted by *self* occurs in the sequence denoted by cr_c , assuming $x \in IVar_d^c$ for some d . However, this cannot be determined statically! Note also that we cannot force the existing objects of a class, say class c , to occur in a particular order in the sequence denoted by cr_c . Our solution to this problem consists essentially of using a second logical expression, of type *Bool*, to describe under which conditions the first expression correctly translates the original one. We will also need a number of logical variables that range over integers, more precisely, over indices in the sequences cr_c . In our example above, the expression x is then translated into the triple $(\langle i \rangle, \text{self} \doteq cr_c \cdot i, iv_x \cdot i)$, where i is some logical integer variable. This is interpreted as follows: Whenever the variable i takes such a value that the Boolean expression $\text{self} \doteq cr_c \cdot i$ is true, then the expression $iv_x \cdot i$ takes the desired value.

The analogue of these transformations in the standard completeness proof is the substitution $[\bar{z}/\bar{x}]$, where \bar{z} is the part of the logical environment which is used to store the part of the state as specified by \bar{x} .

Definition 6.26

We define $l_a^c[\bar{st}] = (\bar{i}, l_{1_{\text{Bool}}}^c, l_{2_a}^c)$ for an arbitrary logical expression l_a^c by induction on the structure of l_a^c . Let ϵ denote the empty sequence. We treat the following cases:

- $x_d^c[\bar{st}] = (\langle i \rangle, \text{self} \doteq cr_c \cdot i, iv_x \cdot i)$
where i is some fresh logical integer variable (it does not occur in \bar{st}).
- $u_d[\bar{st}] = (\epsilon, \text{true}, tv_u)$

- $l[\bar{st}] = (\epsilon, \text{true}, l)$
where $l = \text{nil}, \text{self}, \text{true}, \text{false}, n$, or z .
- $(l_c \cdot x_d^c)[\bar{st}] = (\bar{i} \circ \langle j \rangle, l_1 \wedge l_2 \doteq cr_c \cdot j, iv_x \cdot j)$
where $l_c[\bar{st}] = (\bar{i}, l_1, l_2)$ and $j \notin \bar{i}$.
- $(l_1 + l_2)[\bar{st}] = (\bar{i}, l_{1_1} \wedge l'_{2_1}, l_{1_2} + l'_{2_2})$
where $l_1[\bar{st}] = (\bar{i}_1, l_{1_1}, l_{1_2})$, $l_2[\bar{st}] = (\bar{i}_2, l_{2_1}, l_{2_2})$, $\bar{i} = \bar{i}_1 \circ \bar{j}$, \bar{j} is some sequence of fresh logical integer variables of the same length as \bar{i}_2 , $l'_{2_1} = l_{2_1}[\bar{j}/\bar{i}_2]$, and $l'_{2_2} = l_{2_2}[\bar{j}/\bar{i}_2]$.
- $(\text{if } l_1 \text{ then } l_2 \text{ else } l_3 \text{ fi})[\bar{st}] = (\bar{i}, l_{1_1} \wedge l'_{2_1} \wedge l'_{3_1}, \text{if } l_{1_2} \text{ then } l'_{2_2} \text{ else } l'_{3_2} \text{ fi})$
where $l_1[\bar{st}] = (\bar{i}_1, l_{1_1}, l_{1_2})$, $l_2[\bar{st}] = (\bar{i}_2, l_{2_1}, l_{2_2})$, $l_3[\bar{st}] = (\bar{i}_3, l_{3_1}, l_{3_2})$, $\bar{i} = \bar{i}_1 \circ \bar{j}_2 \circ \bar{j}_3$, \bar{j}_2 and \bar{j}_3 are mutually disjoint sequences of fresh logical variables of the same length as \bar{i}_2 and \bar{i}_3 , respectively, $l'_{2_1} = l_{2_1}[\bar{j}_2/\bar{i}_2]$, $l'_{2_2} = l_{2_2}[\bar{j}_2/\bar{i}_2]$, $l'_{3_1} = l_{3_1}[\bar{j}_3/\bar{i}_3]$, and $l'_{3_2} = l_{3_2}[\bar{j}_3/\bar{i}_3]$.
- $(l_1 \cdot l_2)[\bar{st}] = (\bar{i}, l_{1_1} \wedge l'_{2_1}, l_{1_2} \cdot l'_{2_2})$
where $l_1[\bar{st}] = (\bar{i}_1, l_{1_1}, l_{1_2})$, $l_2[\bar{st}] = (\bar{i}_2, l_{2_1}, l_{2_2})$, $\bar{i} = \bar{i}_1 \circ \bar{j}$, \bar{j} is some sequence of fresh logical integer variables of the same length as \bar{i}_2 , $l'_{2_1} = l_{2_1}[\bar{j}/\bar{i}_2]$, and $l'_{2_2} = l_{2_2}[\bar{j}/\bar{i}_2]$.
- $(l_1 \doteq l_2)[\bar{st}] = (\bar{i}, l_{1_1} \wedge l'_{2_1}, l_{1_2} \doteq l'_{2_2})$
where $l_1[\bar{st}] = (\bar{i}_1, l_{1_1}, l_{1_2})$, $l_2[\bar{st}] = (\bar{i}_2, l_{2_1}, l_{2_2})$, $\bar{i} = \bar{i}_1 \circ \bar{j}$, \bar{j} is some sequence of fresh logical integer variables of the same length as \bar{i}_2 , $l'_{2_1} = l_{2_1}[\bar{j}/\bar{i}_2]$, and $l'_{2_2} = l_{2_2}[\bar{j}/\bar{i}_2]$.

Note that in $l_a^c[\bar{st}] = (\bar{i}, l_1, l_2)$, the expression l_1 describes where the relevant existing objects, with respect to the evaluation of l_a^c , are stored in that part of the logical environment as specified by cr_c , $c \in C$. An object is said to be of relevance with respect to the evaluation of an expression if it requires the values of some variables of this object. The expression l_2 then uses this information to select the relevant values in that part of the logical environment where the values of the variables of the existing objects are stored.

Example 6.27

Consider the expression $z.x.y$, where $z \in LVar_c$, $x \in IVar_c^c$. We have

$$(z.x.y)[\bar{st}] = (\langle i, j \rangle, z \doteq cr_c \cdot i \wedge iv_x \cdot i \doteq cr_c \cdot j, iv_y \cdot j)$$

where i and j are distinct logical integer variables.

Definition 6.28

Next we define the transformation $P^c[\bar{st}]$ for an arbitrary assertion P^c by induction on the structure of P^c . We treat the following cases:

- $l_{\text{Bool}}^c[\bar{st}] = \exists \bar{i}(l_1 \wedge l_2)$
where $l_{\text{Bool}}^c[\bar{st}] = (\bar{i}, l_1, l_2)$.
- $(P_1 \wedge P_2)[\bar{st}] = P_1[\bar{st}] \wedge P_2[\bar{st}], \dots$
- $(\forall z_a P)[\bar{st}] = \forall z_a P[\bar{st}]$,
where $a = d, d^*, d = \text{Int}, \text{Bool}$.
- $(\forall z_c P)[\bar{st}] = \forall z_c (z_c \in cr_c \rightarrow P[\bar{st}])$.
- $(\forall z_a P)[\bar{st}] = \forall z_a (z_a \subseteq cr_c \rightarrow P[\bar{st}])$,
where $a = c^*$.
- $(\exists z_a P)[\bar{st}] = \exists z_a P[\bar{st}]$,
where $a = d, d^*, d = \text{Int}, \text{Bool}$.
- $(\exists z_c P)[\bar{st}] = \exists z_c (z_c \in cr_c \wedge P[\bar{st}])$.
- $(\exists z_a P)[\bar{st}] = \exists z_a (z_a \subseteq cr_c \wedge P[\bar{st}])$,
where $a = c^*$.

Here $l_1 \in l_2$ abbreviates $\exists i(l_1 \doteq l_2 \cdot i)$ and $l_1 \subseteq l_2$ abbreviates $\forall i(l_1 \cdot i \in l_2)$. Note that, although $\text{nil} \in cr_c$, the quantification in $(\forall z_c P)[\bar{st}]$ and $(\exists z_c P)[\bar{st}]$ excludes nil , because quantification always excludes nil .

The following theorem states that the above transformation as applied to assertions preserves truth. It can be seen as an analogue of the substitution lemma of first-order predicate logic.

Theorem 6.29

Let P^c be an arbitrary assertion. Furthermore let σ, δ, ω such that $OK(\sigma, \delta, \omega)$ and $\sigma, \delta, \omega \models \text{init}$. Then:

$$\sigma, \delta, \omega \models P^c \text{ iff } \sigma, \delta, \omega \models P^c[\bar{st}].$$

Proof

The proof proceeds by induction on the structure of P^c . The case that P^c equals l_{Bool}^c is treated as follows: We prove that for every logical expression l_a^c there exists a sequence of integers \bar{n} such that $\sigma, \delta, \omega\{\bar{n}/\bar{i}\} \models l_1$ and that for all such \bar{n} we have $\mathcal{L}[[l_a^c]](\omega)(\delta)(\sigma) = \mathcal{L}[[l_2]](\omega\{\bar{n}/\bar{i}\})(\sigma)$, where $l_a^c[\bar{st}] = (\bar{i}, l_1, l_2)$. This is proved by induction on the structure of l_a^c . \square

6.4 Invariance

In this section we formulate a syntactic criterion for an assertion to be invariant over the execution of an arbitrary program. First we note that not allowing program variables to occur in an assertion does *not* guarantee this invariance property! This is due to the restriction of the range of the quantifiers to existing objects. Consider the following example:

Example 6.30

Let P denote the assertion $\exists z \forall z' (z \doteq z')$, where $z, z' \in LVar_c$ for some class name c . This assertion P expresses that there exists precisely one object of class c . Let $\rho^c = \langle U | x \leftarrow \text{new} \rangle$, U arbitrary and $x \in IVar_c^c$. Then it is not the case that $\models \{P\} \rho^c \{P\}$, because there exist two objects of class c in the output state.

However, the standard technique to prove completeness relies heavily on the invariance of assertions in which no program variables occur. To be able to apply this technique we define the notion of *quantification-restricted* assertions.

Definition 6.31

We define an assertion P^c to be *quantification-restricted* if

$$\begin{aligned}
 P^c &::= l_{\text{Bool}}^c \\
 &\quad \vdots \\
 &\quad | \quad \exists z_a P \mid \forall z_a P \\
 &\quad \quad \text{where } a = d, d^*, d = \text{Int}, \text{Bool} \\
 &\quad | \quad \exists z_c (z_c \in z_{c^*} \wedge P^c) \\
 &\quad | \quad \exists z_{c^*} (z_{c^*} \subseteq z'_c \wedge P^c) \\
 &\quad | \quad \forall z_c (z_c \in z_{c^*} \rightarrow P^c) \\
 &\quad | \quad \forall z_{c^*} (z_{c^*} \subseteq z'_c \rightarrow P^c)
 \end{aligned}$$

Here we assume the variables z_{c^*} and z'_c to be distinct and the assertion P at the right-hand side of the symbol $::=$ to be quantification-restricted.

An important property of such a quantification-restricted assertion is that its truth is not affected by the creation of new objects:

Lemma 6.32

For every quantification-restricted assertion P and every variable v such that $v \notin IVar(P) \cup TVar(P)$ we have $\models P \leftrightarrow P[\text{new}/v]$.

Proof

Induction on the complexity of P . We treat the representative case of $P = \exists z_c (z_c \in$

$z_c \cdot \wedge Q$), assuming the type of the variable v to be c : Now $P[\text{new}/v] = \exists z_c (z_c \in z_c \cdot \wedge Q[\text{new}/v]) \vee (v \in z_c \cdot \wedge Q[v/z_c])[\text{new}/v]$. But as $(v \in z_c \cdot)[\text{new}/v]$ can be easily seen to be equivalent to false the second disjunct will be equivalent to false too. Furthermore we have by the induction hypothesis that $Q[\text{new}/v]$ is equivalent to Q . Putting these observations together gives us the equivalence of P and $P[\text{new}/v]$. The case $P = \forall z_c (z_c \in z_c' \cdot \rightarrow Q)$ is treated analogously. The cases of $P = \exists z_c \cdot (z_c \subseteq z_c' \cdot \wedge Q), \forall z_c \cdot (z_c \subseteq z_c' \cdot \rightarrow Q)$ are slightly more complex due to the complexity of the substitution operations involved, but the reasoning pattern is basically the same. \square

A consequence of this lemma is the following *invariance* property of quantification-restricted assertions:

Theorem 6.33

Let $\rho^c = \langle U|c : S \rangle$ be closed and P^c be a quantification-restricted assertion such that $IVar(P^c) \cap IVar(\rho^c) = \emptyset$ and $TVar(P^c) \cap TVar(\rho^c) = \emptyset$. Then: $\vdash \{P^c\} \rho^c \{P^c\}$.

Proof

The proof proceeds by induction on the complexity of S . We consider the case of $S = v \leftarrow e_0!m(e_1, \dots, e_n)$: Let M be the smallest set such that

- $\rho \in M$,
- if $\rho' = \langle U|c' : v' \leftarrow e_0!m'(e_1', \dots, e_k') \rangle \in M$
then $\rho_i = \langle U|c_i : v_i \leftarrow e_0!m_i(e_1^i, \dots, e_{n_i}^i) \rangle \in M$,
where $v_i \leftarrow e_0!m_i(e_1^i, \dots, e_{n_i}^i)$ or $e_0!m_i(e_1^i, \dots, e_{n_i}^i)$ occurs in S' , S' being the body of the method m' . In the latter case we have $v_i = re_{d_i}$, assuming d_i to be the type of the result expression of m_i .

Let $M = \{\rho_1, \dots, \rho_k\}$, $\rho = \rho_1$, assuming the following notational conventions: $\rho_i = \langle U|c_i : v_i \leftarrow e_0!m_i(e_1^i, \dots, e_{n_i}^i) \rangle \in M$ and $m_i(u_1^i, \dots, u_{n_i}^i) \Leftarrow S_i \uparrow e_i$ occurs in U , $i = 1, \dots, k$. Furthermore, \bar{e}^i denotes the sequence $e_1^i, \dots, e_{n_i}^i$ and \bar{u}^i the sequence $u_1^i, \dots, u_{n_i}^i$. Next we introduce for every class name c a new variable $z_c \cdot$. We let \bar{z} denote a sequence (without repetitions) of these variables and \bar{b} denote the corresponding sequence of the variables $b_c \in BVar$. Finally we put for $i = 1, \dots, k$: $F_i = \{P'\} \rho_i \{P'\}$, where $P' = P^c[\bar{z}/\bar{b}][z_c/\text{self}]$, z_c being a new variable.

Now we have that

$$F_1, \dots, F_k \vdash \{P'\} \langle U|c_i' : S_i \rangle \{P'\}$$

(c_i' being the type of e_0^i). This is established by induction on the complexity of S_i . The only slightly less straightforward case of $S_i = v \leftarrow \text{new}$ is taken care by the previous lemma.

Putting $P_i, Q_i, R_i = P'$ and introducing some logical variable $r_i \notin LVar(P')$ (of the same type as the variable v_i), $i = 1, \dots, k$, and observing that $P'[\bar{e}^i/self, \bar{u}^i][b_{c_i} \circ \langle self \rangle / b_{c_i}] = P'$ we infer by (NMR) that:

$$\vdash \{P'\} \langle U | c'_1 : S_1 \rangle \{P'\}.$$

Next we put $P_1, Q_1 = P'$ and $R_1 = P^c[\bar{z}/\bar{b}]$. We have that:

$$\models P_1[\bar{e}^1/self, \bar{u}^1][self/z_{c_1}][b_{c_1} \circ \langle self \rangle / b_{c_1}] \rightarrow P^c[\bar{z}/\bar{b}]$$

and

$$\models Q_1[\bar{e}^1/self, \bar{u}^1][self/z_{c_1}][b_{c_1} \circ \langle self \rangle / b_{c_1}] \rightarrow R_1[r_1/v_1].$$

Thus applying (MI) (or (MT)) gives us that:

$$\vdash \{P^c[\bar{z}/\bar{b}]\} \rho^c \{P^c[\bar{z}/\bar{b}]\}.$$

Finally an application of the substitution rule gives us the derivability of the correctness formula $\{P^c\} \rho^c \{P^c\}$. \square

6.5 Most general correctness formulae

Now we are able to prove that for an arbitrary $\rho^c = \langle U | c : v \leftarrow e_0!m(e_1, \dots, m_n) \rangle$ the correctness formula $\{init\} \rho^c \{SP_L^c(\rho^c, init)\}$, for some $L \subseteq LVar$, is a most general one in the sense that an arbitrary valid correctness formula can be derived from the proof system which results from adding these correctness formulae as additional axioms. Completeness then follows by establishing the derivability of $\{init\} \rho^c \{SP_L^c(\rho^c, init)\}$, for an arbitrary $\rho^c = \langle U | c : v \leftarrow e_0!m(e_1, \dots, m_n) \rangle$.

But first we need to introduce some new logical variables corresponding to those of \bar{st} . This is necessary because the variables of \bar{st} have a fixed interpretation as specified by the assertion $init$. But every valid correctness formula in which variables of \bar{st} occur, implicitly provides these variables with some possibly different interpretation. To avoid a clash between these different interpretations we must temporarily substitute in the correctness formula, of which we want to establish its derivability, every variable of \bar{st} by some corresponding new variable.

So we introduce for each c fresh logical variables $cr1_c, bl1_c \in LVar_c^*$. For each instance variable $x \in IVar_d$ we introduce the fresh logical variable $iv1_x \in LVar_d^*$, and with each temporary variable $u \in TVar_d$ we associate the fresh logical variable $tv1_u \in LVar_d^*$. We assume again that all these newly introduced logical variables are distinct. We let $\bar{st}1$ denote a sequence (without repetitions) of these variables. We can thus assume that $\bar{st} \cap \bar{st}1 = \emptyset$.

Furthermore we introduce for every temporary variable re_d (defined in the introduction to justify the assumption of the finiteness of the sets C , $IVar$, and $TVar$) a fresh logical variable lre_d . Let \bar{lre} denote a sequence of these logical variables. We will use the variable lre when applying the rule (ES): Applications of this rule will make use of the variable re to store temporarily the result of the expression s . Therefore we have to substitute occurrences of re in the precondition and the postcondition by the corresponding variable lre . We will see later how to restore the original precondition and postcondition after such an application of the rule (ES).

We start with the following lemma stating the derivability of valid correctness formulae about simple assignments.

Lemma 6.34

For an arbitrary program $\rho = \langle U | c : v \leftarrow e \rangle$ we have

$$\models \{P^c\}\rho\{Q^c\} \text{ implies } \vdash \{P^c\}\rho\{Q^c\}.$$

Proof

Let $v = u$, u some temporary variable. (The case of v being an instance variable is treated similarly.) By lemma 5.4 (note that we actually mean here the corresponding lemma for the proof system based on the assertion language with quantification over sequences) and the assumption that $\models \{P^c\}\rho\{Q\}$ it follows that $\models P^c \rightarrow Q^c[e/u]$. So an application of the axiom (SAT) and the consequence rule gives us the derivability of the correctness formula $\{P^c\}\rho\{Q\}$. \square

We have a similar lemma for the creation of new objects:

Lemma 6.35

For an arbitrary program $\rho = \langle U | c : v \leftarrow \text{new} \rangle$ we have

$$\models \{P^c\}\rho\{Q^c\} \text{ implies } \vdash \{P^c\}\rho\{Q^c\}.$$

Proof

Let $v = u$, u some temporary variable. (The case of v being an instance variable is treated similarly.) By lemma 5.18 and the assumption that $\models \{P^c\}\rho\{Q\}$ it follows that $\models P^c \rightarrow Q^c[\text{new}/v]$. So an application of the axiom (NT) and the consequence rule gives us the derivability of the correctness formula $\{P^c\}\rho\{Q\}$. \square

Next we have the following lemma stating the derivability of an arbitrary valid correctness formula about sending messages:

Lemma 6.36

Let $\rho = \langle U|c : v \leftarrow e_0!m(e_1, \dots, e_n) \rangle$ be a closed program. Furthermore let P^c, Q^c and $BVar \subseteq L \subseteq LVar$ (L finite) such that $LVar(P, Q) \subseteq L \setminus \bar{st}I$, and $\bar{st} \cup \bar{st}I \subseteq L$. Then:

$$\models \{P^c\}\rho\{Q^c\} \text{ implies } \{init\}\rho\{SP_L^c(\rho, init)\} \vdash \{P^c\}\rho\{Q^c\}.$$

Proof

Let $P' = P[\bar{st}I/\bar{st}]$ and $Q' = Q[\bar{st}I/\bar{st}]$. Furthermore we introduce the following abbreviation: $P'' = P'[\bar{st}]$. We start with the assumption:

$$\{init\}\rho\{SP_L^c(\rho, init)\}.$$

By theorem 6.33 (note that P'' is quantification-restricted, $IVar(P'') = \emptyset$, and $TVar(P'') = \emptyset$) we have the derivability of the following formula:

$$\{P''\}\rho\{P''\}.$$

Applying the conjunction rule gives us:

$$\{P'' \wedge init\}\rho\{P'' \wedge SP_L^c(\rho, init)\}.$$

We next prove that $\models P'' \wedge SP_L^c(\rho, init) \rightarrow Q'$:

Let $\sigma, \delta, \omega \models P'' \wedge SP_L^c(\rho, init)$. So there exist a state σ_0 and an *osi* f such that

- $f(\sigma) = \mathcal{P}[\rho](\gamma)(\delta')(\sigma_0)$, γ arbitrary,
- $\sigma_0, \delta', \omega' \models init$,

where $\delta' = f(\delta)$ and $\omega' = f(\omega) \upharpoonright L$.

By theorem 6.19 we have that $f(\sigma), f(\delta), f(\omega) \models P''$. It is not difficult to check that $LVar(P'') \subseteq L$, so we have $f(\sigma), \delta', \omega' \models P''$. Furthermore we have that $\models \{\neg P''\}\rho\{\neg P''\}$ (by theorem 6.33 we have $\vdash \{\neg P''\}\rho\{\neg P''\}$, so the truth of the above correctness formula follows from the soundness of the proof system). It follows that $\sigma_0, \delta', \omega' \models P''$. By theorem 6.29, note that $\sigma_0, \delta', \omega' \models init$, we then infer $\sigma_0, \delta', \omega' \models P'$. By the soundness of the substitution rule (SR) we have that $\models \{P\}\rho\{Q\}$ implies the truth of the correctness formula $\{P'\}\rho\{Q'\}$. So we infer that $f(\sigma), \delta', \omega' \models Q'$. But as $LVar(Q') \subseteq L$ we have $f(\sigma), \delta', f(\omega) \models Q'$. Finally an application of theorem 6.19 gives us the desired result $\sigma, \delta, \omega \models Q'$.

Now we return to our main argument. By the consequence rule we thus infer:

$$\{P'' \wedge \text{init}\} \rho \{Q'\}.$$

Next we apply the initialization rule (IR1):

$$\{(P'' \wedge \text{init})[\bar{u}/\bar{t}v]\} \rho \{Q'\},$$

where \bar{u} is a sequence of all the temporary variables and $\bar{t}v$ denotes the corresponding sequence of logical variables tv_u , $u \in \bar{u}$. Now we use the elimination rule (ER2):

$$\{\exists \bar{z}'(P'' \wedge \text{init})[\bar{u}/\bar{t}v]\} \rho \{Q'\},$$

where \bar{z}' is a sequence of the logical variables $\{cr_c, bl_c : c \in C\}$ and $\{iv_x : x \in IVar\}$. Note that instead of initializing the variables $\bar{t}v$ we could also eliminate them by rule (ER1). However, applying the rule (ER1) would require some additional notational machinery in order to deal with the extra case of nil.

Next we prove $\models P' \rightarrow \exists \bar{z}'(P'' \wedge \text{init})[\bar{u}/\bar{t}v]$: Let σ, δ, ω be such that $OK(\sigma, \delta, \omega)$ and $\sigma, \delta, \omega \models P'$. It is not difficult to see that there exists an ω' such that ω' differs from ω only with respect to the variables of \bar{st} and $\sigma, \delta, \omega' \models \text{init}$. As $LVar(P') \cap \bar{st} = \emptyset$ we have $\sigma, \delta, \omega' \models P'$. Applying theorem 6.29 then gives us $\sigma, \delta, \omega' \models P'[\bar{st}]$. For every temporary variable u we have $\sigma_{(3)}(u) = \omega'(tv_u)$, so we infer $\sigma, \delta, \omega' \models (P'' \wedge \text{init})[\bar{u}/\bar{t}v]$. So we conclude $\sigma, \delta, \omega \models \exists \bar{z}'(P'' \wedge \text{init})[\bar{u}/\bar{t}v]$.

We thus have by the consequence rule:

$$\{P'\} \rho \{Q'\}.$$

Finally an application of the substitution rule finishes the proof. Note that since $LVar(P^c, Q^c) \cap \bar{st}1 = \emptyset$, we have that $P'[\bar{st}/\bar{st}1] = P^c$ and $Q'[\bar{st}/\bar{st}1] = Q^c$, so we get

$$\{P\} \rho \{Q\}.$$

□

We next have lemmas 6.38 and 6.39 stating the derivability of valid correctness formulae about statements $S = s$, where s is a side-effect expression. In these two lemmas we make use of the following lemma:

Lemma 6.37

Let $\rho = \langle U|c : s \rangle$ and $\rho' = \langle U|c : re \leftarrow s \rangle$ be restricted programs (see definition 6.12). We then have for arbitrary assertions P and Q that

$$\models \{P\} \rho \{Q\} \text{ implies } \models \{P'\} \rho' \{Q'\},$$

where $P' = P[lre/re]$ and $Q' = Q[lre/re]$.

Proof

Let $\sigma, \delta, \omega \models P'$ and $\sigma' = \mathcal{P}[\rho'](\gamma)(\delta)(\sigma)$. We have that $\sigma' = \sigma''\{\beta/re\}$, with $\langle \sigma'', \beta \rangle = \mathcal{Z}[s](\gamma')(\delta)(\sigma)$, $\gamma' = \mathcal{U}[U](\gamma)$. As $re \notin TVar(s)$ (ρ being restricted) we have $\langle \sigma_1, \beta \rangle = \mathcal{Z}[s](\gamma')(\delta)(\sigma_0)$, with $\sigma_1 = \sigma''\{\omega(lre)/re\}$ and $\sigma_0 = \sigma\{\omega(lre)/re\}$. This being intuitively clear we feel justified in stating it without a proof. Now, as $\sigma, \delta, \omega \models P'$ we have that $\sigma_0, \delta, \omega \models P$. So from $\models \{P\}\rho\{Q\}$ we then infer $\sigma_1, \delta, \omega \models Q$, or, equivalently, $\sigma'', \delta, \omega \models Q'$. Finally, as $re \notin TVar(Q')$, we conclude that $\sigma', \delta, \omega \models Q'$. \square

Lemma 6.38

Let $\rho = \langle U|c : s \rangle$, where $s = e, \text{new}$. Furthermore let P, Q such that $LVar(P, Q) \cap \bar{lre} = \emptyset$. Then:

$$\models \{P\}\rho\{Q\} \text{ implies } \vdash \{P\}\rho\{Q\}.$$

Proof

Let $P' = P[lre/re]$ and $Q' = Q[lre/re]$, where lre and re are of the same type as the expression s . By lemma 6.37 we have $\models \{P'\}\rho'\{Q'\}$, where $\rho' = \langle U|c : re \leftarrow s \rangle$. By lemma 6.34, in case $s = e$, and lemma 6.35, if $s = \text{new}$, we then have

$$\vdash \{P'\}\rho'\{Q'\}.$$

So by rule (ES) it follows that

$$\vdash \{P'\}\rho\{Q'\}.$$

Furthermore we have $\models \{lre \doteq re\} \langle U|c : re' \leftarrow s \rangle \{lre \doteq re\}$. So again by lemmas 6.34 and 6.35 we have

$$\vdash \{lre \doteq re\} \langle U|c : re' \leftarrow s \rangle \{lre \doteq re\}.$$

Applying again the rule (ES) then gives

$$\vdash \{lre \doteq re\}\rho\{lre \doteq re\}.$$

Next we apply the conjunction rule

$$\vdash \{lre \doteq re \wedge P'\}\rho\{lre \doteq re \wedge Q'\}.$$

Now $\models (lre \doteq re \wedge Q') \rightarrow Q$ and $\models P \rightarrow (\exists lre P'' \vee P''[\text{nil}/lre])$, where $P'' = lre \doteq re \wedge P'$. (Note that $lre \notin LVar(P)$.) So applying first the consequence rule for Q , then the elimination rule (ER1) (note that $lre \notin LVar(Q)$), and finally the consequence rule for P , gives us the derivability of

$$\vdash \{P\}\rho\{Q\}.$$

□

We have a similar lemma for valid correctness formulae about a program ρ of the form $\langle U|c : e_0!m(e_1, \dots, e_n) \rangle$.

Lemma 6.39

Let $\rho = \langle U|c : e_0!m(e_1, \dots, e_n) \rangle$ be a closed program. Furthermore let P, Q , and $BVar \subseteq L \subseteq LVar$ (L finite) such that $LVar(P, Q) \subseteq L \setminus (\bar{st}l \cup \bar{l}re)$, $\bar{st} \cup \bar{st}l \cup \bar{l}re \subseteq L$. Then we have

$$\models \{P\}_\rho \{Q\} \text{ implies } \{init\}_{\rho'} \{SP_L(\rho', init)\} \vdash \{P\}_\rho \{Q\},$$

where $\rho' = \langle U|c : re_d \leftarrow e_0!m(e_1, \dots, e_n) \rangle$, assuming the type of the result expression of m to be d .

Proof

Let $P' = P[lre_d/re_d]$ and $Q' = Q[lre_d/re_d]$. An application of lemma 6.37 gives us $\models \{P'\}_{\rho'} \{Q'\}$ (remember that ρ is assumed to be restricted). By lemma 6.36 we have

$$\{init\}_{\rho'} \{SP_L(\rho', init)\} \vdash \{P'\}_{\rho'} \{Q'\}.$$

Applying next the rule (ES) gives us

$$\{init\}_{\rho'} \{SP_L(\rho, init)\} \vdash \{P'\}_{\rho} \{Q'\}.$$

By theorem 6.33 (observe that $re_d \notin TVar(\rho)$) we have the derivability of the formula

$$\vdash \{lre_d \doteq re_d\}_\rho \{lre_d \doteq re_d\}.$$

So by an application of the conjunction rule we have

$$\{init\}_{\rho} \{SP_L(\rho, init)\} \vdash \{P' \wedge lre_d \doteq re_d\}_\rho \{Q' \wedge lre_d \doteq re_d\}.$$

Now we have $\models (Q' \wedge lre_d \doteq re_d) \rightarrow Q$. Furthermore for $P'' = P' \wedge lre_d \doteq re_d$ we have $\models P \rightarrow (\exists lre_d P'' \vee P''[nil/lre_d])$ (note that $lre_d \notin LVar(P)$). So first applying the consequence rule for Q , then the elimination rule (ER1) (note that $lre_d \notin LVar(Q)$), and finally the consequence rule for P finishes the proof. □

Next we have the following main theorem of this section stating the derivability of an arbitrary valid correctness formula using as additional axioms the correctness formulae of the form $\{init\}_\rho \{SP_L(\rho, init)\}$, where $\rho = \langle U|c : v \leftarrow e_0!m(e_1, \dots, e_n) \rangle$.

Theorem 6.40

Let $\rho = \langle U|c : S \rangle$ be a closed program. Furthermore let P^c, Q^c , and $BVar \subseteq L \subseteq LVar$ (L finite) such that $LVar(P^c, Q^c) \subseteq L \setminus (\bar{st}l \cup \bar{l}re)$, $\bar{st} \cup \bar{st}l \cup \bar{l}re \subseteq L$. Then:

$$\models \{P^c\}_\rho \{Q^c\} \text{ implies } F_1, \dots, F_n \vdash \{P^c\}_\rho \{Q^c\},$$

where $F_i = \{init\}\rho_i\{SP_L^{c_i}(\rho_i, init)\}$, $\rho_i = \langle U|c_i : v_i \leftarrow s_i \rangle$, with s_1, \dots, s_n being all the send-expressions occurring in S such that $v_i \leftarrow s_i$ occurs in S or $v_i = re_{d_i}$ and s_i occurs as a statement in S . Here d_i is assumed to be the type of s_i .

Proof

The proof proceeds by induction on the complexity of S .

$S = v \leftarrow s$: Depending on the structure of s , by one of the lemmas 6.34, 6.35, 6.36.

$S = s$: Depending on the structure of s , by one of the lemmas 6.38, 6.39.

$S = S_1; S_2$:

Let $L^- = L \setminus (s\bar{t}l \cup l\bar{r}e)$. We have by lemma 6.22

$$\models \{P^c\}\rho_1\{SP_{L^-}(\rho_1, P^c)\},$$

and

$$\models \{SP_{L^-}(\rho_1, P^c)\}\rho_2\{SP_{L^-}(\rho_2, SP_{L^-}(\rho_1, P^c))\},$$

where $\rho_i = \langle U|c : S_i \rangle$. By the induction hypothesis we have

$$F_1, \dots, F_n \vdash \{P^c\}\rho_1\{SP_{L^-}(\rho_1, P^c)\},$$

and

$$F_1, \dots, F_n \vdash \{SP_{L^-}(\rho_1, P^c)\}\rho_2\{SP_{L^-}(\rho_2, SP_{L^-}(\rho_1, P^c))\}.$$

It thus suffices to prove that $\models SP_{L^-}(\rho_2, SP_{L^-}(\rho_1, P^c)) \rightarrow Q^c$: An application of the rule for sequential composition (SC) and the consequence rule then gives us the desired result.

So suppose that $\sigma, \delta, \omega \models SP_{L^-}(\rho_2, SP_{L^-}(\rho_1, P^c))$, with $OK(\sigma, \delta, \omega)$. By theorem 6.21 there exist a state σ_0 and an *osi* f such that

- $f(\sigma) = \mathcal{P}[\rho_2](\gamma)(\delta')(\sigma_0)$, γ arbitrary,
- $\sigma_0, \delta', \omega' \models SP_{L^-}(\rho_1, P^c)$,

where $\delta' = f(\delta)$ and $\omega' = f(\omega) \downarrow L^-$.

Now $\sigma_0, \delta', \omega' \models SP_{L^-}(\rho_1, P^c)$ in turn implies that there exist a state σ'_0 and an *osi* g such that

- $g(\sigma_0) = \mathcal{P}[\rho_1](\gamma)(\delta'')(\sigma'_0)$, γ arbitrary,

$$\bullet \sigma'_0, \delta'', \omega'' \models P^c,$$

where $\delta'' = g(\delta')$ and $\omega'' = g(\omega') \downarrow L^-$.

To relate these computations of ρ_1 and ρ_2 we apply corollary C.8 of appendix C: There exists an *osi* h such that $h^c \downarrow \sigma_0^{(c)} = g^c \downarrow \sigma_0^{(c)}$, for every c , and $h(f(\sigma)) = \mathcal{P}[\rho_2](\gamma)(g(\delta'))(g(\sigma_0))$, where γ is arbitrary.

Since $g(\delta') = \delta''$ it follows that $h(f(\sigma)) = \mathcal{P}[\rho](\gamma)(\delta'')(\sigma'_0)$, with γ arbitrary. So by $\sigma'_0, \delta'', \omega'' \models P^c$ and $\models \{P^c\} \rho \{Q^c\}$ we infer $h(f(\sigma)), \delta'', \omega'' \models Q^c$.

Now note that $OK(\sigma_0, \delta')$. So we have $h(\delta'_{(1)}) = g^c(\delta'_{(1)}) = \delta''_{(1)}$ and $h(\delta'_{(2)(c)}) = g(\delta'_{(2)(c)}) = \delta''_{(2)(c)}$, for every c . Thus we infer that $\delta'' = h(\delta') = h(f(\delta))$. Moreover for $z \in L^-$ we have $h(f(\omega(z))) = h(\omega'(z)) = g(\omega'(z)) = \omega''(z)$. Note that the second identity is justified by $OK(\sigma_0, \delta', \omega')$. So by theorem 6.19 and the fact that $LVar(Q^c) \subseteq L^-$ we conclude $\sigma, \delta, \omega \models Q^c$.

S is if ... fi: Straightforward.

$S = \text{while } e \text{ do } S_1 \text{ od}$:

In order to deal with this case we construct a loop invariant R as follows. Let $L^- = L \setminus (s\bar{l}l \cup l\bar{r}e)$ and $L^+ = L^- \cup \{z_u, z_{u'}\}$, where z_u and $z_{u'}$ are some new logical integer variables. We define $P' = P[z_u, z_{u'}/u, u']$ and $Q' = Q[z_u, z_{u'}/u, u']$. Let $\rho' = \langle U | c : \text{while } e \wedge u < u' \text{ do } S_1; u \leftarrow u + 1 \text{ od} \rangle$. Furthermore let $R' = SP_{L^+}(\rho', P' \wedge u \doteq 0)$ and define $R = \exists z R'[z, z/u, u']$, where $z \in LVar_{Int}$ is a new variable. Note that $LVar(R) \subseteq L^+$. Furthermore we have $\models \{P'\} \rho \{Q'\}$ (note that $u, u' \notin TVar(\rho)$, ρ being restricted).

We have $\models P' \rightarrow R$:

Let $\sigma, \delta, \omega \models P'$, with $OK(\sigma, \delta, \omega)$. We prove that for $\omega' = \omega\{0/z\}$ we have $\sigma, \delta, \omega' \models R'[z, z/u, u']$. Now $\sigma, \delta, \omega' \models R'[z, z/u, u']$ iff $\sigma', \delta, \omega \models R'$ by a straightforward extension of lemma 5.4 (note that $z \notin Exp$), where $\sigma' = \sigma\{0, 0/u, u'\}$ (note that $z \notin LVar(R')$). Because $u, u' \notin TVar(P')$ we have $\sigma', \delta, \omega \models P' \wedge u \doteq 0$. Furthermore it is easy to see that $\sigma' = \mathcal{P}[\rho'](\gamma)(\delta)(\sigma')$, with γ arbitrary. Finally, as $LVar(P') \subseteq L^+$ we have by theorem 6.21 $\sigma', \delta, \omega \models R'$.

Next we prove $\models R \wedge \neg e \rightarrow Q'$:

Let $\sigma, \delta, \omega \models R \wedge \neg e$. So let $\alpha \in \mathbb{N}$ such that $\sigma', \delta, \omega \models R'$, where $\sigma' = \sigma\{\alpha, \alpha/u, u'\}$. So there exist f, σ_0 such that

- $f(\sigma') = \mathcal{P}[\rho'](\gamma)(\delta')(\sigma_0)$, γ arbitrary,
- $\sigma_0, \delta', \omega' \models P' \wedge u \doteq 0$,

where $\delta' = f(\delta)$ and $\omega' = f(\omega) \downarrow L^+$. Now $u, u' \notin TVar(e)$ so $\sigma, \delta, \omega \models \neg e$ implies $\sigma', \delta, \omega \models \neg e$. By theorem 6.19 we have $f(\sigma'), \delta', f(\omega) \models \neg e$. So from $LVar(e) = \emptyset$ it follows that $f(\sigma'), \delta', \omega' \models \neg e$. From this it is not difficult to derive that $f(\sigma') = \mathcal{P}[\rho](\gamma)(\delta')(\sigma'_0)$, where $\sigma'_0 = \sigma_0\{\alpha, \alpha/u, u'\}$. Now as $u, u' \notin TVar(P')$ it follows that $\sigma'_0, \delta', \omega' \models P'$. So by $\models \{P'\} \rho \{Q'\}$ we have $f(\sigma'), \delta', \omega' \models Q'$. By $LVar(Q') \subseteq L^+$ and theorem 6.19 we have $\sigma', \delta, \omega \models Q'$. So that from $u, u' \notin TVar(Q')$ we finally conclude $\sigma, \delta, \omega \models Q'$.

Finally, we have $\models \{R \wedge e\} \rho_1 \{R\}$, where $\rho_1 = \langle U|c : S_1 \rangle$:

Let $\sigma_0, \delta, \omega \models R \wedge e$, with $OK(\sigma_0, \delta, \omega)$, and $\sigma_1 = \mathcal{P}[\rho_1](\gamma)(\delta)(\sigma_0)$, with γ arbitrary. Let $\alpha \in N$ such that $\sigma'_0, \delta, \omega \models R'$, where $\sigma'_0 = \sigma_0\{\alpha, \alpha/u, u'\}$. So there exist f, σ such that

- $f(\sigma'_0) = \mathcal{P}[\rho'](\gamma)(\delta')(\sigma)$, γ arbitrary,
- $\sigma, \delta', \omega' \models P' \wedge u \doteq 0$,

where $\delta' = f(\delta)$ and $\omega' = f(\omega) \downarrow L^+$.

So we have the following situation:

$$\begin{array}{ccc} \sigma_0, \delta & \xrightarrow{\rho_1} & \sigma_1 \\ | & & \\ \sigma'_0 & & \\ | & & \\ \sigma, \delta' & \xrightarrow{\rho'} & f(\sigma'_0) \end{array}$$

Here $\sigma, \delta \xrightarrow{\rho} \sigma'$ should be interpreted as $\sigma' = \mathcal{P}[\rho](\gamma)(\delta)(\sigma)$, γ arbitrary. We have $\sigma'_0, \delta, \omega \models e$ because $u, u' \notin TVar(e)$. So by theorem 6.19 and $LVar(e) = \emptyset$ we infer $f(\sigma'_0), \delta', \omega' \models e$. Now let $\sigma'_1 = \sigma_1\{\alpha, \alpha/u, u'\}$. It then follows that $\sigma'_1 = \mathcal{P}[\rho_1](\gamma)(\delta)(\sigma'_0)$. We now have the following situation:

$$\begin{array}{ccc} \sigma_0, \delta & \xrightarrow{\rho_1} & \sigma_1 \\ | & & | \\ \sigma'_0, \delta & \xrightarrow{\rho_1} & \sigma'_1 \\ | & & \\ \sigma, \delta' & \xrightarrow{\rho'} & f(\sigma'_0) \end{array}$$

An application of corollary C.8 then gives us an *osi* g such that $g^c \downarrow \sigma_0^{(c)} = f^c \downarrow \sigma'_0^{(c)}$ for every c , and $g(\sigma'_1) = \mathcal{P}[\rho_1](\gamma)(g(\delta))(f(\sigma'_0))$, with γ arbitrary. Note that from

$OK(\sigma'_0, \delta)$ it then follows that $g(\delta) = f(\delta) = \delta'$. Finally, we thus have reached the following situation:

$$\begin{array}{ccccc}
 \sigma_0, \delta & & \xrightarrow{\rho_1} & & \sigma_1 \\
 | & & & & | \\
 \sigma'_0, \delta & & \xrightarrow{\rho_1} & & \sigma'_1 \\
 | & & & & | \\
 \sigma, \delta' & \xrightarrow{\rho'} & f(\sigma'_0), \delta' & \xrightarrow{\rho_1} & g(\sigma'_1)
 \end{array}$$

Now it follows that for $\sigma_2 = \sigma\{\alpha + 1/u'\}$ and $\sigma_3 = g(\sigma'_1)\{\alpha + 1, \alpha + 1/u, u'\}$ we have $\sigma_3 = \mathcal{P}[\rho'](\gamma)(\delta')(\sigma_2)$, with γ arbitrary. (Of course this can be proved formally, but as the intuition behind a formal proof is quite obvious, the main idea being simply that the temporary variable u counts the number of loops, we think we are justified in omitting such a proof.) Now $\sigma, \delta', \omega' \models P'$, $u, u' \notin TVar(P')$, so $\sigma_2, \delta', \omega' \models P'$, from which in turn it follows by lemma 6.22 that $\sigma_3, \delta', \omega' \models R'$. So we infer $g(\sigma'_1), \delta', \omega' \models R$. Now $LVar(R) \subseteq L^+$ and for $z \in L^+$ we have $g(\omega(z)) = f(\omega(z)) = \omega'(z)$ (the first identity follows from $OK(\sigma'_0, \omega)$) so we have $g(\sigma'_1), \delta', g(\omega) \models R$. It follows by an application of theorem 6.19 that $\sigma'_1, \delta, \omega \models R$. Finally, as we have $u, u' \notin TVar(R)$ we conclude $\sigma_1, \delta, \omega \models R$.

Now by $\models \{R \wedge e\}\rho_1\{R\}$ it follows that $\models \{R'' \wedge e\}\rho_1\{R''\}$ (note that $u, u' \notin TVar(\rho_1)$), where $R'' = R[u, u'/z_u, z_{u'}]$. As $LVar(R'') \subseteq L^-$ we can apply the induction hypothesis:

$$F_1, \dots, F_n \vdash \{R'' \wedge e\}\rho_1\{R''\}.$$

By theorem 6.33 we have

$$\vdash \{z_u \doteq u \wedge z_{u'} \doteq u'\}\rho_1\{z_u \doteq u \wedge z_{u'} \doteq u'\}.$$

Furthermore we have $\models (R'' \wedge z_u \doteq u \wedge z_{u'} \doteq u') \rightarrow R$ and $R \rightarrow (R'' \wedge z_u \doteq u \wedge z_{u'} \doteq u')[z_u, z_{u'}/u, u']$ (note that $u, u' \notin TVar(R)$). So applying the conjunction rule, the consequence rule for the postcondition, the initialization rule (IR2), and the consequence rule for the precondition gives us

$$F_1, \dots, F_n \vdash \{R \wedge e\}\rho_1\{R\}.$$

From an application of the rule (W) and the consequence rule, using the truth of the implications $P' \rightarrow R$ and $R \wedge \neg e \rightarrow Q'$, it then follows that:

$$F_1, \dots, F_n \vdash \{P'\}\rho\{Q'\}.$$

Now again by an application of theorem 6.33 and the conjunction rule we have

$$F_1, \dots, F_n \vdash \{P' \wedge z_u \doteq u \wedge z_{u'} \doteq u'\}\rho\{Q' \wedge z_u \doteq u \wedge z_{u'} \doteq u'\}.$$

We have $\models (Q' \wedge z_u \doteq u \wedge z_{u'} \doteq u') \rightarrow Q$ and $\models P \rightarrow (P' \wedge z_u \doteq u \wedge z_{u'} \doteq u')[u, u'/z_u, z_{u'}]$. So applying first the consequence rule for Q , then the initialization rule (IR1), and finally the consequence rule for P gives us the desired result. \square

6.6 The context switch

In this subsection we prove the derivability of the correctness formula $\{init\}\rho\{SP_L^c(\rho, init)\}$, for $\rho = \langle U|c : v \leftarrow e_0!m(e_1, \dots, e_n) \rangle$ closed and $BVar \subseteq L \subseteq LVar$ such that $\bar{st} \cup \bar{sl} \cup \bar{lr} \subseteq L$. From now on until the end of this section unless stated otherwise we assume ρ and L to be fixed. We want to apply the rule (NMR) and theorem 6.40. To apply the rule (NMR) we need the following definition:

Definition 6.41

Let M be the smallest set such that

- $\rho \in M$,
- if $\rho' = \langle U|c' : v' \leftarrow e_0!m'(e'_1, \dots, e'_k) \rangle \in M$
then $\rho_i = \langle U|c_i : v_i \leftarrow e_0!m_i(e_1^i, \dots, e_{n_i}^i) \rangle \in M$,
where $v_i \leftarrow e_0!m_i(e_1^i, \dots, e_{n_i}^i)$ or $e_0!m_i(e_1^i, \dots, e_{n_i}^i)$ occurs in S' as a statement
(in this latter case we have $v_i = re_{d_i}$, assuming d_i to be the type of the result
expression of m_i), S' being the body of the method m' .

Let $M = \{\rho_1, \dots, \rho_k\}$, $\rho = \rho_1$, assuming the following notational conventions: $\rho_i = \langle U|c_i : v_i \leftarrow e_0!m_i(e_1^i, \dots, e_{n_i}^i) \rangle \in M$, and $m_i(u_1^i, \dots, u_{n_i}^i) \Leftarrow S_i \upharpoonright e_i$ occurs in U , $i = 1, \dots, k$. We let \bar{e}^i denote the sequence $e_0^i, \dots, e_{n_i}^i$. Furthermore let \bar{u} be a sequence of all the temporary variables, and let the formal parameters of the method m_i be denoted by \bar{u}^i .

We start with a sketch of the proof strategy. To apply theorem 6.40 and the rule (NMR) we have to define assertions P_i, Q_i , $i = 1, \dots, k$, such that $LVar(P_i, Q_i) \subseteq L \setminus (\bar{st} \cup \bar{lr})$, and

$$\models \left\{ P_i \wedge \bigwedge_j v_j^i \doteq \text{nil} \wedge \text{self} \notin b_{c_i'} \right\} \langle U|c_i' : S_i \rangle \left\{ Q_i[e_i/r_i] \right\}, \quad (6.1)$$

where $\bar{v}^i = \bar{u} \setminus \bar{u}^i$ and c_i' is the type of e_0^i ,

$$\models \text{init} \rightarrow P_i[\bar{e}^i/\text{self}, \bar{u}^i][\bar{g}^i/\bar{z}^i][b_{c_i} \circ \langle \text{self} \rangle / b_{c_i}] \quad (6.2)$$

and

$$\models Q_i[\bar{e}^i/\text{self}, \bar{u}^i][\bar{g}^i/\bar{z}^i][b_{c_i} \circ \langle \text{self} \rangle / b_{c_i}] \rightarrow SP_L^c(\rho_i, \text{init})[r_i/v_i], \quad (6.3)$$

for some sequence of expressions \bar{g}^i and corresponding sequence of logical variables \bar{z}^i . Here r_i for $i = 1, \dots, k$ is a logical variable of the same type as v_i . By 6.1 an application of theorem 6.40 then gives us

$$F'_1, \dots, F'_k \vdash \left\{ P_i \wedge \bigwedge_j v_j^i \doteq \text{nil} \wedge \text{self} \notin b_{c_i'} \right\} \langle U|c_i' : S_i \rangle \left\{ Q_i[e_i/r_i] \right\}$$

where

$$F'_i = \{init\} \rho_i \{SP_L^{c_i}(\rho_i, init)\}.$$

Furthermore by an application of the consequence rule, using (6.2), we have $F_i \vdash F'_i$ where

$$F_i = \{P_i[\bar{e}^i/self, \bar{u}^i][\bar{g}^i/\bar{z}^i][b_{c_i} \circ \langle self \rangle / b_{c_i}]\} \rho_i \{SP_L^{c_i}(\rho_i, init)\}.$$

So we have

$$F_1, \dots, F_k \vdash \left\{ P_i \wedge \bigwedge_j v_j^i \doteq nil \wedge self \notin b_{c_i} \right\} \langle U | c_i' : S_i \rangle \{ Q_i[e_i/r_i] \}.$$

An application of (NMR) plus (MI) or (MT) and the consequence rule, using (6.2) and (6.3), then concludes the proof.

We start with the considering equations (6.2) and (6.3): We define a substitution which neutralizes the context switch. To do so we first introduce some new logical variables.

Definition 6.42

We associate with $u \in \bar{u}$ a new logical variable $tv2_u$ of the same type and with each $c \in C$ a new logical variable id_c . We define $\overline{tv2}$ to be the sequence of logical variables $tv2_u$ corresponding to the sequence \bar{u} . Finally let \bar{id}^i , $i = 1, \dots, k$, denote the sequence consisting of the variable id_{c_i} followed by the elements of $\overline{tv2}$.

We have the following lemma about the neutralizing capacity of the substitution $[\bar{id}^i/self, \bar{u}]$ with respect to the context switch:

Lemma 6.43

For any $i \in \{1, \dots, k\}$ and every assertion $P \in Ass^{c_i}$ we have

$$P^{c_i}[\bar{id}^i/self, \bar{u}][\bar{e}^i/self, \bar{u}^i] = P^{c_i}[\bar{id}^i/self, \bar{u}].$$

Proof

Straightforward induction on the complexity of P^{c_i} . □

Note that the substitution $[\bar{id}^i/self, \bar{u}]$ transforms the assertion P^{c_i} into an assertion in Ass^c for arbitrary c . Furthermore it is easy to see that if $LVar(P) \cap \bar{id}^i = \emptyset$ then $\models P^{c_i} \leftrightarrow P^{c_i}[\bar{id}^i/self, \bar{u}][\bar{f}/\bar{id}^i]$, where \bar{f} denotes the sequence consisting of the expression $self$ followed by the elements of \bar{u} . Note that in general we do *not* have that P^{c_i} is syntactically equal to $P^{c_i}[\bar{id}^i/self, \bar{u}][\bar{f}/\bar{id}^i]$, as is shown by the following example:

Example 6.44

Take for $P^{c_i} = x \doteq z.y$, where $z \notin \bar{id}^i$. We have $P^{c_i}[\bar{id}^i/self, \bar{u}] = id_{c_i}.x \doteq z.y$ and $(id_{c_i}.x \doteq z.y)[\bar{f}/\bar{id}^i] = self.x \doteq z.y$.

Next we consider the substitution $[b_{c_i} \circ \langle self \rangle / b_{c_i}]$. It is not difficult to see that for every assertion P^{c_i} we have

$$\models (P^{c_i}[bl_{c_i}/b_{c_i}] \wedge b_{c_i} = bl_{c_i} \circ \langle self \rangle)[b_{c_i} \circ \langle self \rangle / b_{c_i}] \rightarrow P^{c_i}.$$

But note that we do *not* have the other way around! However, as $\models init \rightarrow bl_{c_i} = b_{c_i}$, we *do* have

$$\models init \rightarrow ((init[bl_{c_i}/b_{c_i}] \wedge b_{c_i} = bl_{c_i} \circ \langle self \rangle)[b_{c_i} \circ \langle self \rangle / b_{c_i}]).$$

To summarize the argument above we introduce the following definition:

Definition 6.45

For any $i \in \{1, \dots, k\}$ and any assertion $P \in Ass^{c_i}$ we define its *reverse context switch* $\mathbf{R}(P^{c_i})$ as follows:

$$\mathbf{R}(P^{c_i}) = (P^{c_i}[bl_{c_i}/b_{c_i}] \wedge b_{c_i} = bl_{c_i} \circ \langle self \rangle)[\bar{id}^i/self, \bar{u}]$$

We have the following lemma about this reverse context switch:

Lemma 6.46

For any $i \in \{1, \dots, k\}$ and every assertion $P \in Ass^{c_i}$ we have

$$\models \mathbf{R}(P^{c_i})[\bar{e}^i/self, \bar{u}][\bar{f}/\bar{id}^i][b_{c_i} \circ \langle self \rangle / b_{c_i}] \rightarrow P^{c_i}.$$

and if $\models P^{c_i} \rightarrow b_{c_i} \doteq bl_{c_i}$ then

$$\models P^{c_i} \rightarrow \mathbf{R}(P^{c_i})[\bar{e}^i/self, \bar{u}][\bar{f}/\bar{id}^i][b_{c_i} \circ \langle self \rangle / b_{c_i}].$$

Here $\bar{f} = self, \bar{u}$.

Proof

Clear from the above. □

So at this stage candidates for $P_i, Q_i, i = 1, \dots, k$, satisfying equations (6.2) and (6.3) are the assertions $\mathbf{R}(init)$ and $\mathbf{R}(SP_L^{c_i}(\rho_i, init)[r_i/v_i])$, $i = 1, \dots, k$. We now proceed by analyzing equation (6.1). Suppose we are given that for some P and Q we have $\models \{P\}\rho_i\{Q\}$. In general we do *not* have

$$\models \left\{ \mathbf{R}(P) \wedge \bigwedge_j v_j^i \doteq nil \wedge self \notin b_{c_i'} \right\} \langle U | c_i' : S_i \rangle \left\{ \mathbf{R}(Q') [e_i/r_i] \right\},$$

where $Q' = Q[r_i/v_i]$. This is because it is possible that the object executing S_i is not the object which is sent the message and furthermore nothing is said about the values of the formal parameters. So we add to $\mathbf{R}(P)$ the information $\text{self} \doteq e_0^i[\bar{id}^i/\text{self}, \bar{u}]$ and $u_j^i \doteq e_j^i[\bar{id}^i/\text{self}, \bar{u}]$, $j = 1, \dots, n_i$. We have the following lemma:

Lemma 6.47

$$\models (f_j^i \doteq (e_j^i[\bar{id}^i/\text{self}, \bar{u}]))[\bar{e}^i/\text{self}, \bar{u}][\bar{f}/\bar{id}^i]$$

where $\bar{f}^i = \text{self}$, \bar{u}^i and $\bar{f} = \text{self}$, \bar{u} .

Proof

Easy. □

Note that from lemma 6.46 and lemma 6.47 it follows that for every P^{c_i} such that $\models P^{c_i} \rightarrow b_{c_i} \doteq bl_{c_i}$ we have

$$\models P \rightarrow \left(\mathbf{R}(P) \wedge \bigwedge_j f_j^i \doteq (e_j^i[\bar{id}^i/\text{self}, \bar{u}]) \right) [\bar{e}^i/\text{self}, \bar{u}][\bar{f}/\bar{id}^i][b_{c_i} \circ \langle \text{self} \rangle / b_{c_i}].$$

Now we are ready for the following lemma which shows how to transform a valid correctness formula about sending a message into a valid formula about the execution of the body of the message by the receiver:

Lemma 6.48

For any $i \in \{1, \dots, k\}$ and every $P, Q \in \text{Ass}^{c_i}$ such that $\models \{P\} \rho_i \{Q\}$ we have

$$\models \left\{ P' \wedge \bigwedge_j v_j^i \doteq \text{nil} \wedge \text{self} \notin b_{c_i'} \right\} \langle U | c_i' : S_i \rangle \left\{ Q'[e_i/r_i] \right\},$$

where $P' = \mathbf{R}(P) \wedge \bigwedge_j f_j^i \doteq (e_j^i[\bar{id}^i/\text{self}, \bar{u}])$ and $Q' = \mathbf{R}(Q[r_i/v_i])$, with r_i a fresh logical variable of the same type as v_i . Here $\bar{v}^i = \bar{u} \setminus \bar{u}^i$.

Proof

Let $\sigma, \delta, \omega \models P' \wedge \bigwedge_j v_j^i \doteq \text{nil} \wedge \text{self} \notin b_{c_i'}$, for σ, δ, ω such that $OK(\sigma, \delta, \omega)$, and $\sigma' = \mathcal{P}[\langle U | c_i' : S_i \rangle](\gamma)(\delta)(\sigma)$, with γ arbitrary and $\sigma' \neq \perp$.

We define $\sigma_1 = \sigma\{\omega(tv2_u)/u\}_{u \in \bar{u}}$ and $\delta'_{(1)} = \omega(id_{c_i})$, $\delta'_{(2)(c)} = \delta_{(2)(c)}$ for every c .

It follows from lemma 5.28 that $\sigma_1, \delta', \omega \models P[bl_{c_i}/b_{c_i}] \wedge b_{c_i} = bl_{c_i} \circ \langle \text{self} \rangle$.

Next we define δ'' as follows: $\delta''_{(1)} = \delta'_{(1)}$, $\delta''_{(2)(c_i)} = \delta'_{(2)(c_i)} \setminus \{\omega(id_{c_i})\}$, and $\delta''_{(2)(c)} = \delta'_{(2)(c)}$ for any $c \neq c_i$. Furthermore we put $\omega_1 = \omega\{\omega(bl_{c_i})/b_{c_i}\}$. It then follows that $OK(\sigma_1, \delta'', \omega_1)$ and $\sigma_1, \delta'', \omega_1 \models P$.

$$\begin{array}{ccc}
\sigma, \delta, \omega \models P' & & \sigma', \delta, \omega \models Q'[e_i/r_i] \\
\Downarrow & & \Uparrow \\
\sigma_1, \delta', \omega \models P[bl_{c_i}/b_{c_i}] \wedge b_{c_i} = bl_{c_i} \circ \langle \text{self} \rangle & \sigma'', \delta', \omega_3 \models Q[bl_{c_i}, r_i/b_{c_i}, v_i] \wedge b_{c_i} = bl_{c_i} \circ \langle \text{self} \rangle & \\
\Downarrow & & \Uparrow \\
\sigma_1, \delta'', \omega_1 \models P & \Rightarrow & \sigma_2, \delta'', \omega_1 \models Q
\end{array}$$

Figure 4: The structure of the proof of lemma 6.48.

Let on the other hand $\sigma'' = \sigma' \{ \omega(tv2_u)/u \}_{u \in \bar{u}}$ and

$$\sigma_2 = \begin{cases} \sigma'' \{ \beta/v_i \} & \text{if } v_i \in TVar \\ \sigma'' \{ \beta/\omega(id_{c_i}), v_i \} & \text{if } v_i \in IVar, \end{cases}$$

where $\beta = \mathcal{E}[e_i](\delta)(\sigma')$ (remember that e_i is the result expression of the method m_i). Now from $\sigma, \delta, \omega \models \bigwedge_j f_j^i = (e_j^i[\bar{id}^i/\text{self}, \bar{u}])$ it follows from lemma 5.28 that $\delta_{(1)} = \mathcal{L}[e_0^i[\bar{id}^i/\text{self}, \bar{u}]](\omega)(\delta)(\sigma) = \mathcal{L}[e_0^i](\omega)(\delta'')(\sigma_1)$ and $\sigma_{(3)}(u_j^i) = \mathcal{L}[e_j^i](\omega)(\delta'')(\sigma_1)$. Furthermore from $\sigma, \delta, \omega \models \text{self} \notin b_{c_i}'$ it in turn follows that $\delta_{(1)} \not\subseteq \delta_{(2)(c_i)'}$. Now putting this together with the assumption that $\sigma' = \mathcal{P}[\langle U|c_i' : S_i \rangle](\gamma)(\delta)(\sigma)$, using $\sigma, \delta, \omega \models \bigwedge_j v_j^i \doteq \text{nil}$, enables one to infer that $\sigma_2 = \mathcal{P}[\rho_i](\gamma)(\delta'')(\sigma_1)$.

Furthermore we are given that $\models \{P\}\rho_i\{Q\}$ so from $\sigma_1, \delta'', \omega_1 \models P$ and $\sigma_2 = \mathcal{P}[\rho_i](\gamma)(\delta'')(\sigma_1)$ we infer that $\sigma_2, \delta'', \omega_1 \models Q$. Now let $\omega_2 = \omega_1 \{ \beta/r_i \}$. It then follows by lemma 5.8 that $\sigma'', \delta'', \omega_2 \models Q[r_i/v_i]$. Next we note that as $\omega_2(b_{c_i}) = \omega_1(b_{c_i}) = \omega(bl_{c_i})$ we have $\sigma'', \delta', \omega_3 \models Q[r_i, bl_{c_i}/v_i, b_{c_i}]$, where $\omega_3 = \omega \{ \beta/r_i \}$.

From $\sigma, \delta, \omega \models \mathbf{R}(P)$ we infer that $\omega(b_{c_i}) = \omega(bl_{c_i}) \circ \langle \omega(id_{c_i}) \rangle$. But $\omega(id_{c_i}) = \delta'_{(1)}$ so we have $\sigma'', \delta', \omega_3 \models Q[r_i, bl_{c_i}/v_i, b_{c_i}] \wedge b_{c_i} = bl_{c_i} \circ \langle \text{self} \rangle$.

Now an application of lemma 5.28 gives us $\sigma', \delta, \omega_3 \models \mathbf{R}(Q[r_i/v_i])$. From this in turn it follows that $\sigma', \delta, \omega \models Q'[e_i/r_i]$. \square

Now we want to apply lemma 6.48 taking $init$ for P and $SP_L^{c_i}(\rho_i, init)$ for Q . Note that by lemma 6.22 we have $\models \{init\}\rho_i\{SP_L^{c_i}(\rho_i, init)\}$. Now taking for P_i the assertion $\mathbf{R}(init) \wedge \bigwedge_j f_j^i \doteq (\bar{e}_j^i[\bar{z}/\text{self}, \bar{u}])$ and for Q_i the assertion $\mathbf{R}(SP_L^{c_i}(\rho_i, init)[r_i/v_i])$ we have by lemma 6.46 and lemma 6.47 that equations (6.2) and (6.3) are satisfied. However since in the assertions P_i and Q_i new logical variables occur which are not contained in L , we must apply theorem 6.40 for $F_i = \{init\}\rho_i\{SP_{L^+}^{c_i}(\rho_i, init)\}$, where $L^+ = L \cup \{id_c : c \in C\} \cup \{tv2_u : u \in \bar{u}\}$. But to apply the rule (NMR) we then have to take for Q_i the assertion $\mathbf{R}(SP_{L^+}^{c_i}(\rho_i, init)[r_i/v_i])$. An application

of (NMR) and (MI) or (MT) would then give us the derivability of the correctness formula $\{init\} \rho \{SP_{L^+}^c(\rho, init)\}$. However, as $\models SP_{L^+}^c(\rho, init) \rightarrow SP_L^c(\rho, init)$ (use $LVar(init) \subseteq L \subseteq L^+$), we have by an application of the consequence rule the derivability of $\{init\} \rho \{SP_L^c(\rho, init)\}$.

But there is one problem we did not discuss yet. As $\bar{st}1 \cup \bar{lre} \subseteq LVar(SP_{L^+}^{ci}(\rho_i, init))$ we can not apply theorem 6.40! This problem is solved as follows: First we define $L^- = L^+ \setminus (\bar{st}1 \cup \bar{lre})$. Next we define the following abbreviation:

Definition 6.49

Let $Subs(\bar{lre}, \bar{st}1, \bar{cr})$ abbreviate the assertion:

$$\bigwedge_c (cr1_c \subseteq cr_c \wedge bl1_c \subseteq cr_c \wedge lre_c \in cr_c \wedge \bigwedge_{d \in C} \bigwedge_{x \in IVar_d^c} iv1_x \subseteq cr_d \wedge \bigwedge_{u \in TVar_c} tv1_u \in cr_c).$$

The assertion $Subs(\bar{lre}, \bar{st}1, \bar{cr})$ states that all the objects which are denoted by a variable of \bar{lre} or $\bar{st}1$, or which occur in a sequence denoted by some variable of $\bar{st}1$, are stored in the corresponding variable of \bar{cr} . We have the following proposition:

Proposition 6.50

Let $P_i = \mathbf{R}(init) \wedge \bigwedge_j f_j^i \doteq (e_j^i[id^i/self, \bar{u}])$, $Q_i^- = \mathbf{R}(SP_{L^-}^{ci}(\rho_i, init)[r_i/v_i])$ and $Q_i^+ = \mathbf{R}(SP_{L^+}^{ci}(\rho_i, init)[r_i/v_i])$. We have

$$\models P_i \wedge Subs(\bar{lre}, \bar{st}1, \bar{cr}) \leftrightarrow P_i$$

and

$$\models Q_i^-[e_i/r_i] \wedge Subs(\bar{lre}, \bar{st}1, \bar{cr}) \rightarrow Q_i^+[e_i/r_i].$$

Proof

The first assertion follows immediately from the fact that the assertion $init$ (and so the assertion $\mathbf{R}(init)$) implies the assertion $\forall z_c (z_c \in cr_c)$, for every c .

Now we prove the second assertion. Let $\sigma, \delta, \omega \models Q_i^-[e_i/r_i] \wedge Subs(\bar{lre}, \bar{st}1, \bar{cr})$. For $\omega_1 = \omega\{\mathcal{E}[e_i](\delta)(\sigma)/r_i\}$, we then have $\sigma, \delta, \omega_1 \models Q_i^- \wedge Subs(\bar{lre}, \bar{st}1, \bar{cr})$.

Next we define $\sigma' = \sigma\{\omega_1(tv2_u)/u\}_{u \in \bar{u}}$, and $\delta'_{(1)} = \omega_1(id_{c_i})$, $\delta'_{(2)(c)} = \delta_{(2)(c)}$, for every c . It then follows by lemma 5.28 that: $\sigma', \delta', \omega_1 \models SP_{L^-}^{ci}(\rho_i, init)[r_i, bl_{c_i}/v_i, b_{c_i}] \wedge b_{c_i} = bl_{c_i} \circ \langle self \rangle \wedge Subs(\bar{lre}, \bar{st}1, \bar{cr})$.

For $\omega_2 = \omega_1\{\omega_1(bl_{c_i})/b_{c_i}\}$ and $\delta''_{(1)} = \delta'_{(1)}$, for $c \neq c_i$: $\delta''_{(2)(c)} = \delta'_{(2)(c)}$, otherwise: $\delta''_{(2)(c)} = \delta'_{(2)(c)} \setminus \delta'_{(1)}$, we have $\sigma', \delta'', \omega_2 \models SP_{L^-}^{ci}(\rho_i, init)[r_i/v_i] \wedge Subs(\bar{lre}, \bar{st}1, \bar{cr})$.

Next, let

$$\sigma'' = \begin{cases} \sigma' \{ \omega_2(r_i) / \delta'_{(1)}, v_i \} & \text{if } v_i \in IVar \\ \sigma' \{ \omega_2(r_i) / v_i \} & \text{if } v_i \in TVar. \end{cases}$$

It follows that $\sigma'', \delta'', \omega_2 \models SP_{L-}^{c_i}(\rho_i, init) \wedge Subs(\bar{l}re, \bar{s}t1, \bar{c}r)$.

So by theorem 6.21 there exist f and σ_0 such that:

- $f(\sigma'') = \mathcal{P}[\rho](\gamma)(f(\delta''))(\sigma_0)$, with γ arbitrary,
- $\sigma_0, f(\delta''), \omega' \models init$,

where $\omega' = f(\omega_2) \downarrow L^-$. Let $\sigma_1 = f(\sigma'')$. Now by theorem 6.19 we have that $\sigma_1, f(\delta''), f(\omega_2) \models Subs(\bar{l}re, \bar{s}t1, \bar{c}r)$. So from $\{cr_c : c \in C\} \subseteq L^-$ and the compatibility of ω' and σ_0 we then infer the compatibility of $f(\omega_2) \downarrow L^+$ and σ_0 . Let $\omega'' = f(\omega_2) \downarrow L^+$. We have that $\sigma_0, f(\delta''), \omega'' \models init$, so we have $\sigma'', \delta'', \omega_2 \models SP_{L+}^{c_i}(\rho_i, init)$. From this it follows, by “reversing” the part of the above argument which led to the statement $\sigma'', \delta'', \omega_2 \models SP_{L-}^{c_i}(\rho_i, init)$, that $\sigma, \delta, \omega \models Q_i^+[e_i/r_i]$. \square

Now we are ready for the following theorem.

Theorem 6.51

Let the program $\rho = \langle U | c : v \leftarrow e_0!m(e_1, \dots, e_n) \rangle$ be closed and let $BVar \subseteq L \subseteq LVar$ such that $\bar{s}t \cup \bar{s}t1 \cup \bar{l}re \subseteq L$. Then we have

$$\vdash \{init\} \rho \{SP_L^c(\rho, init)\}.$$

Proof

Let $P_i = \mathbf{R}(init) \wedge \bigwedge_j f_j^i \doteq (e_j^i[id^i/self, \bar{u}])$, $Q_i^- = \mathbf{R}(SP_{L-}^{c_i}(\rho_i, init)[r_i/v_i])$ and $Q_i^+ = \mathbf{R}(SP_{L+}^{c_i}(\rho_i, init)[r_i/v_i])$. Now by lemma 6.22 we get

$$\models \{init\} \rho_i \{SP_{L-}^{c_i}(\rho_i, init)\}$$

So we have, by lemma 6.48,

$$\models \{P_i \wedge \bigwedge_j v_j^i \doteq nil \wedge self \notin b_{c_i'}\} \langle U | c_i' : S_i \rangle \{Q_i^-[e_i/r_i]\}.$$

An application of theorem 6.40 then gives us (note that the restrictions on the logical variables are satisfied)

$$F'_1, \dots, F'_k \vdash \{P_i \wedge \bigwedge_j v_j^i \doteq nil \wedge self \notin b_{c_i'}\} \langle U | c_i' : S_i \rangle \{Q_i^-[e_i/r_i]\},$$

where

$$F'_i = \{init\} \rho_i \{SP_{L+}^{c_i}(\rho_i, init)\}.$$

Now by lemma 6.46 and lemma 6.47 an application of the consequence rule gives us $F_i \vdash F'_i$ where

$$F_i = \{P_i[\bar{e}^i/self, \bar{u}^i][\bar{f}/\bar{id}^i][b_{c_i} \circ \langle self \rangle / b_{c_i}]\} \rho_i \{SP_{L+}^{c_i}(\rho_i, init)\}.$$

So we have

$$F_1, \dots, F_k \vdash \left\{ P_i \wedge \bigwedge_j v_j^i \doteq nil \wedge self \notin b_{c_i} \right\} \langle U|c'_i : S_i \rangle \{Q_i^-[e_i/r_i]\}.$$

By theorem 6.33 we have

$$\vdash \{Subs(\bar{l}re, \bar{s}t1, \bar{c}r)\} \langle U|c'_i : S_i \rangle \{Subs(\bar{l}re, \bar{s}t1, \bar{c}r)\}.$$

So by the conjunction rule we infer

$$F_1, \dots, F_m \vdash \left\{ P_i \wedge \bigwedge_j v_j^i \doteq nil \wedge self \notin b_{c_i} \wedge Subs(\bar{l}re, \bar{s}t1, \bar{c}r) \right\} \langle U|c'_i : S_i \rangle \{Q_i^-[e_i/r_i] \wedge Subs(\bar{l}re, \bar{s}t1, \bar{c}r)\}.$$

By proposition 6.50 an application of the consequence rule gives us

$$F_1, \dots, F_m \vdash \left\{ P_i \wedge \bigwedge_j v_j^i \doteq nil \wedge self \notin b_{c_i} \right\} \langle U|c'_i : S_i \rangle \{Q_i^+[e_i/r_i]\}.$$

We now can apply rule (NMR), making use of lemma 6.46, yielding the derivability of the correctness formula:

$$\left\{ P_1 \wedge \bigwedge_j v_j^1 \doteq nil \wedge self \notin b_{c_1} \right\} \langle U|c'_1 : S_1 \rangle \{Q_1^+[e_1/r_1]\}.$$

Applying next (MI) or (MT) gives us the derivability of

$$\{P_1[\bar{e}^1/self, \bar{u}^1][\bar{f}/\bar{z}^1][b_{c_1} \circ \langle self \rangle / b_{c_1}]\} \rho_1 \{SP_{L+}(\rho_1, init)\}.$$

So an application of the consequence rule (the assertion *init* by lemma 6.46 implies the precondition, and $\models SP_{L+}(\rho_1, init) \rightarrow SP_L(\rho_1, init)$) gives us the desired result (note that $\rho_1 = \rho$ by definition)

$$\vdash \{init\} \rho \{SP_L^c(\rho, init)\}.$$

□

We conclude with the completeness theorem:

Theorem 6.52

Let $\rho^c = \langle U|c : S \rangle$ be a closed program. We have for an arbitrary correctness formula $\{P^c\}_{\rho^c}\{Q^c\}$:

$$\models \{P^c\}_{\rho^c}\{Q^c\} \text{ implies } \vdash \{P^c\}_{\rho^c}\{Q^c\}.$$

Proof

Let P' and Q' result from substituting for every variable of $\bar{st}l$ and $\bar{l}re$ a corresponding new variable (new with respect to the sets $LVar(P^c, Q^c)$, $\bar{st}l$, $\bar{st}l$, $\bar{l}re$). Let $L \subseteq LVar$ (L finite) be such that $BVar \subseteq L$, $LVar(P', Q') \subseteq L$ and $\bar{st}l \cup \bar{st}l \cup \bar{l}re \subseteq L$. By the soundness of the substitution rule we have $\models \{P'\}_{\rho^c}\{Q'\}$, so applying theorem 6.40 gives us

$$F_1, \dots, F_n \vdash \{P'\}_{\rho^c}\{Q'\},$$

where $F_i = \{init\}_{\rho_i}\{SP_L^i(\rho_i, init)\}$, $\rho_i = \langle U|c_i : v_i \leftarrow e_0^i!m_i(e_1^i, \dots, e_{n_i}^i) \rangle$ and $e_0^i!m_i(e_1^i, \dots, e_{n_i}^i)$ $i = 1, \dots, n$, are all the send-expressions occurring in S , and if such an expression $e_0^i!m_i(e_1^i, \dots, e_{n_i}^i)$ occurs in S as a statement we have that $v_i = re_{d_i}$, assuming d_i to be the type of the result expression of m_i . By theorem 6.51 we have the derivability of F_i , so we infer that $\vdash \{P'\}_{\rho^c}\{Q'\}$. Finally an application of the substitution rule gives us the derivability of $\{P^c\}_{\rho^c}\{Q^c\}$. \square

7 Conclusions

In the previous sections we have given a proof system for SPOOL that fulfills the requirements we have listed in the introduction:

- The only possible operations on object references (pointers) are testing for equality and dereferencing.
- In each state of the system only the existing objects play a role in assertions about that state.

In fact, we have given even *two* proof systems fulfilling these requirements: one with recursively defined predicates and one with the ability to reason about finite sequences of objects.

The technique which we have given for computing the weakest precondition for an assignment with respect to a given postcondition, a generalized version of substitution, seems very powerful. Especially the fact that is possible to do this for a new

assignment, in the situation that it is not possible to mention the newly created object in the state before the statement, is a little bit surprising.

The proof rule for message passing, incorporating the passing of parameters and result, context switching, and the constancy of the variables of the sending object, is a very complex rule. It seems to work fine for our proof system, but its properties have not yet been studied extensively enough. It would be interesting to see whether the several things that are handled in one rule could be dealt with by a number of different, simpler rules.

We have proved completeness for the proof system based on the assertion language containing quantification over finite sequences using the standard techniques (see [3], for example). But how to apply these techniques to the proof system based on recursive predicates remains an open problem.

Therefore we must conclude that there is still some work to be done on these issues. In addition, in the present proof systems the protection properties of object are not reflected very well. While in the programming language it is not possible for one object to access the internal details (variables) of another one, in the assertion language this is allowed. In order to improve this it might be necessary to develop a system in which an object presents some abstract view of its behaviour to the outside world. Perhaps techniques developed to deal with abstract data types are useful here.

Finally it is clear that the work on SPOOL is meant as a preparation for the study of POOL, the parallel language. In the following two chapters of this thesis we show how to combine a system like the one presented here with the known techniques for reasoning about parallel programs.

References

- [1] Pierre America: *Definition of the programming language POOL-T*. ESPRIT project 415A, Doc. No. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, September 1985.
- [2] Pierre America: *A proof theory for a sequential version of POOL*. ESPRIT project 415A, Doc. No. 188, Philips Research Laboratories, Eindhoven, the Netherlands.
- [3] Krzysztof R. Apt: *Ten years of Hoare logic: a survey — part I*. ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp. 431-483.
- [4] J.W. de Bakker: *Mathematical Theory of Program Correctness*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

- [5] Herbert B. Enderton: *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [6] Adele Goldberg, David Robson: *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, 1983.
- [7] Joseph M. Morris: *Assignment and linked data structures*. Manfred Broy, Gunther Schmidt (eds.): *Theoretical Foundations of Programming Methodology*. Reidel, 1982, pp. 35–41.
- [8] Dana S. Scott: *Identity and existence in intuitionistic logic*. M.P. Fourman, C.J. Mulvey, D.S. Scott (eds.): *Applications of Sheaves*. Proceedings, Durham 1977, Springer-Verlag, 1979, pp. 660–696 (Lecture Notes in Mathematics 753).
- [9] Joseph R. Shoenfield: *Mathematical Logic*. Addison-Wesley, 1967.
- [10] John V. Tucker, Jeffery I. Zucker: *Program Correctness over Abstract Data Types, with Error-State Semantics*. CWI Monograph Series, Vol. 6, Centre for Mathematics and Computer Science/North-Holland, 1988.

A A generalisation of the rule (MR)

In this section we show that in the recursion rule (MR), as introduced in definition 5.33 and adapted in definition 6.4, we can replace U^- by U itself, thus allowing nested applications of (MR) to the same methods. Let (NMR) denote the recursion rule resulting from (MR) by replacing all occurrences of U^- by U . Furthermore let \vdash denote the derivability using (NMR) (\vdash denotes derivability using (MR)). We have the following theorem:

Theorem A.1

For every correctness formula F we have $\vdash F$ iff $\vdash F$.

Proof

\Rightarrow : We prove that if $F_1, \dots, F_n \vdash F$ then $F_1, \dots, F_n \vdash F$ by induction on the length of the derivation. We treat the case that the last rule applied is (NMR). So let the following be an instance of (NMR):

$$\frac{\tilde{F}_1, \dots, \tilde{F}_n \quad F_1, \dots, F_n \vdash F'_1, \dots, F'_n}{F'_1}$$

where $F'_1 = F$. Let U be the unit occurring in this application of (NMR). We may assume without loss of generality that all the methods declared by U are specified by one of the F_i . (Otherwise, let $\{\rho_1, \dots, \rho_k\}$, where $\rho_i = \langle U | c_i : v_i \leftarrow e_0^i ! m_i(e_1^i, \dots, e_{n_i}^i) \rangle$,

be all the send statements occurring in U . Now simply add to F_1, \dots, F_n for $i = 1, \dots, k$, $G_i = \{\text{true}\} \rho_i \{\text{true}\}$, and note that

$$G_1, \dots, G_k \vdash \{\text{true}\} \langle U | c'_i : S_i \rangle \{\text{true}\}$$

where c'_i is the type of e_0^i and S_i denotes the body of m_i .) We shall prove by induction on the number of applications of (NMR) in the derivation $F_1, \dots, F_n \vdash F'_1, \dots, F'_n$ that for some $\bar{H}_1, \dots, \bar{H}_k, H_1, \dots, H_k, H'_1, \dots, H'_k$ such that for $1 \leq i \leq k$

$$\frac{H'_i \quad \bar{H}_i}{H_i}$$

is an instance of (MI) or (MT), we have:

$$\bar{F}_1, \dots, \bar{F}_n, \bar{H}_1, \dots, \bar{H}_k \vdash \bar{F}'_1, \dots, \bar{F}'_n, \bar{H}'_1, \dots, \bar{H}'_k$$

where, for $G = \{P\} \langle U | c : S \rangle \{Q\}$, \bar{G} denotes $\{P\} \langle E | c : S \rangle \{Q\}$, E being the empty unit. Having proved this we apply (MR) thus yielding $\vdash F'_1 (= F)$. Here we go:

Induction basis: Assume that no application of (NMR) occurs in the derivation $F_1, \dots, F_n \vdash F'_1, \dots, F'_n$. So we have that $F_1, \dots, F_n \vdash^- F'_1, \dots, F'_n$, where \vdash^- denotes derivability from \vdash without (MR). It is not difficult to see that it suffices to prove by induction on the length of the derivation that for an arbitrary correctness formula G if $F_1, \dots, F_n \vdash^- G$ then for some $\bar{H}_1, \dots, \bar{H}_k, H_1, \dots, H_k, H'_1, \dots, H'_k$ we have

$$\bar{F}_1, \dots, \bar{F}_n, \bar{H}_1, \dots, \bar{H}_k \vdash \bar{G}, \bar{H}'_1, \dots, \bar{H}'_k,$$

where for $i = 1, \dots, k$

$$\frac{H'_i \quad \bar{H}_i}{H_i}$$

is an instance of (MI) or (MT). We treat the only interesting case that the last rule applied is an instance of (MI) or (MT). So suppose $F_1, \dots, F_n \vdash^- G', \bar{G}$, where

$$\frac{G' \quad \bar{G}}{G}$$

is an instance of (MI) or (MT). Now by the induction hypothesis we know that for some $\bar{H}_1, \dots, \bar{H}_k, H_1, \dots, H_k, H'_1, \dots, H'_k$:

$$\bar{F}_1, \dots, \bar{F}_n, \bar{H}_1, \dots, \bar{H}_k \vdash \bar{G}', \bar{H}'_1, \dots, \bar{H}'_k,$$

such that for $i = 1, \dots, k$

$$\frac{H'_i \quad \bar{H}_i}{H_i}$$

is an instance of (MI) or (MT). Now let $\bar{H}_{k+1} = \bar{G}$, $H_{k+1} = G$, and $H'_{k+1} = G'$. We then have that

$$\bar{F}_1, \dots, \bar{F}_n, \bar{H}_1, \dots, \bar{H}_{k+1} \vdash \bar{G}, \bar{H}'_1, \dots, \bar{H}'_{k+1}.$$

Induction step: Let for $i = 1, \dots, m$

$$\frac{\tilde{G}_1^i, \dots, \tilde{G}_{n_i}^i \quad G_1^i, \dots, G_{n_i}^i \vdash G_1^{i'}, \dots, G_{n_i}^{i'}}{G_1^{i'}}$$

be all the applications of (NMR) in the derivation $F_1, \dots, F_n \vdash F'_1, \dots, F'_n$ such that

$$F_1, \dots, F_n, G_1^{1'}, \dots, G_1^{m'} \vdash F'_1, \dots, F'_n.$$

By the same induction argument as used in the basis step above we have for some $\tilde{H}_1, \dots, \tilde{H}_k, H_1, \dots, H_k, H'_1, \dots, H'_k$ (such that for $i = 1, \dots, k$

$$\frac{H'_i \quad \tilde{H}_i}{H_i}$$

is an instance of (MI) or (MT)) that

$$F_1, \dots, F_n, H_1, \dots, H_k, G_1^{1'}, \dots, G_1^{m'} \vdash F'_1, \dots, F'_n, H'_1, \dots, H'_k,$$

where \vdash denotes derivability from \vdash minus the rules (MR), (MI), and (MT). Now, applying the induction hypothesis gives us for $i = 1, \dots, m$: $\tilde{H}_1^i, \dots, \tilde{H}_{k_i}^i, H_1^i, \dots, H_{k_i}^i, H_1^{i'}, \dots, H_{k_i}^{i'}$ such that

$$\tilde{G}_1^i, \dots, \tilde{G}_{n_i}^i, \tilde{H}_1^i, \dots, \tilde{H}_{k_i}^i \vdash \tilde{G}_1^{i'}, \dots, \tilde{G}_{n_i}^{i'}, \tilde{H}_1^{i'}, \dots, \tilde{H}_{k_i}^{i'}.$$

Now it follows by a straightforward induction on the length of the derivation

$$F_1, \dots, F_n, H_1, \dots, H_k, G_1^{1'}, \dots, G_1^{m'} \vdash F'_1, \dots, F'_n, H'_1, \dots, H'_k$$

that

$$\mathcal{F} \cup \bigcup_i \mathcal{G}_i \cup \bigcup_i \mathcal{H}_i \vdash \mathcal{F}' \cup \bigcup_i \mathcal{G}'_i \cup \bigcup_i \mathcal{H}'_i$$

where

- $\mathcal{F} = \{\bar{F}_1, \dots, \bar{F}_{n+k}\}, F_{n+i} = H_i, i = 1, \dots, k,$
- $\mathcal{F}' = \{\bar{F}', \dots, \bar{F}'_{n+k}\}, F'_{n+i} = H'_i, i = 1, \dots, k,$
- $\mathcal{G}_i = \{\tilde{G}_1^i, \dots, \tilde{G}_{n_i}^i\}, 1 \leq i \leq m,$
- $\mathcal{G}'_i = \{\tilde{G}_1^{i'}, \dots, \tilde{G}_{n_i}^{i'}\}, 1 \leq i \leq m,$
- $\mathcal{H}_i = \{\tilde{H}_1^i, \dots, \tilde{H}_{k_i}^i\}, 1 \leq i \leq m,$
- $\mathcal{H}'_i = \{\tilde{H}_1^{i'}, \dots, \tilde{H}_{k_i}^{i'}\}, 1 \leq i \leq m.$

\Leftarrow : This is proved in a similar way as the other direction. □

B Expressibility

In this section we show how to formulate the assertion $SP_L^c(\rho^c, P^c)$ in our assertion language, for an arbitrary closed program ρ^c , $BVar \subseteq L \subseteq LVar$ (L finite), such that $LVar(P^c) \subseteq LVar$.

As in section 6 we assume the sets C , $IVar$, and $TVar$ to be finite.

B.1 Coding mappings

Assumption B.1

We assume the existence of the following *coding* mappings:

- For every instance variable or temporary variable $v \in ITVar$ we have $[v] \in \mathbf{N}$, and for an arbitrary program ρ we have $[\rho] \in \mathbf{N}$.
- For every $d \in C^+$, $[\cdot]_d \in \mathbf{O}_\perp^d \rightarrow \mathbf{N}$ denotes an injection such that $[\perp]_d = 0$. In addition, we assume that the function $[\cdot]_{Int}$ is surjective.
- For every state $\sigma \in \Sigma$ such that $OK(\sigma)$, $[\sigma] \in \mathbf{N}$.
- For every context $\delta^c \in \Delta^c$: $[\delta^c] \in \mathbf{N}$.

Furthermore we assume that the mappings $[\cdot]_{Int}$ and $[\cdot]_{Bool}$ are definable in our assertion language. That is, we regard the following function symbols as abbreviations for assertions that are expressible in our assertion language:

- $Ic(n) = m$ (mnemonic: *integer coding*) iff $[n]_{Int} = m$.
- $Bc(b) = m$ (mnemonic: *Boolean coding*) iff $[b]_{Bool} = m$.
- $Id(n) = m$ (mnemonic: *integer decoding*) iff $[m]_{Int} = n$.

To be precise, with the first assumption above we mean that there is an assertion $Ic(z_1) = z_2$, where z_1 and z_2 are integer logical variables, such that for every $\sigma \in \Sigma, \delta \in \Delta, \omega \in \Omega$ with $OK(\sigma, \delta, \omega)$ we have

$$\sigma, \delta, \omega \models Ic(z_1) = z_2 \quad \text{iff} \quad [\omega(z_1)]_{Int} = \omega(z_2).$$

In fact from now on for every $c \in C$ and $\alpha \in \mathbf{O}^c$ we identify $[\alpha]_c$ with α . So we assume $\mathbf{O}^c \subseteq \mathbf{N}$.

In the same way we assume the following predicates and functions to be expressible in our assertion language:

- $E^c(n, m)$ (mnemonic: *exists*) iff there exist a $\sigma \in \Sigma$ and $\alpha \in \sigma^{(c)}$ such that $[\alpha]_c = n$ and $[\sigma] = m$.
- $A^c(n) = m$ (mnemonic: *active*) iff there exists a $\delta \in \Delta^c$ such that $[\delta] = n$ and $[\delta_{(1)}]_c = m$.
- $B^c(n, m)$ (mnemonic: *blocked*) iff there exist a $\delta \in \Delta$ and an $\alpha \in \delta_{(2)(c)}$ such that $[\alpha]_c = n$ and $[\delta] = m$.
- $Val_d^c(k, l, m) = n$ (mnemonic: *value*) iff there exist $\sigma \in \Sigma$, $\alpha \in \sigma^{(c)}$, and $x_d^c \in IVar_d^c$ such that $[\alpha]_c = k$, $[x_d^c] = l$, $[\sigma] = m$, and $[\sigma(\alpha)(x_d^c)]_d = n$.
- $Val_d(l, m) = n$ iff there exist a $\sigma \in \Sigma$ and a $u_d \in TVar_d$ such that $[u_d] = l$, $[\sigma] = m$, and $[\sigma(u_d)]_d = n$.
- $T^c(n, m, l, k)$ (mnemonic: *transforms*) iff there exist a closed $\rho \in Prog^c$, $\delta \in \Delta^c$, and $\sigma, \sigma' \in \Sigma$ such that $OK(\sigma, \delta)$, $[\rho] = n$, $[\delta] = m$, $[\sigma] = l$, $[\sigma'] = k$, and $\sigma' = \mathcal{P}^c[\rho](\gamma)(\delta)(\sigma)$ (where γ is arbitrary).

The above assumptions may appear quite implausible at first sight, but they can be justified by Church's Thesis, which states that every function or relation that can be effectively calculated is recursive, together with the (mathematical) fact that every recursive function is representable in the standard Peano theory of natural numbers and therefore it is certainly definable in our assertion language. (For a discussion of these issues, see [5] or [9].)

B.2 Arithmetizing Truth

To express the strongest postcondition we have to arithmetize the truth of an assertion in a state. More precisely, we will define a translation which transforms an arbitrary assertion into an assertion in which no instance variables or temporary variables occur. The idea of this translation is similar to the one given in the definitions 6.26 and 6.28. But instead of transforming an assertion into an assertion referring to a sequence of logical variables used to store the state, we now transform it into an assertion referring to the *code* of a state. This is necessary to be able to use the predicates of assumption B.1, in particular the predicate T .

To get started we introduce some new variables: Let \overline{bij} denote a sequence of some variables $bij^c \in LVar_{c^*}$, $c \in C$. We shall use these variables to store the essential parts of the bijections that constitute an *osi* (see definition 6.17). The way in which this is done will be made precise in definition B.3, but here we can already explain how the \overline{bij} can be used as a kind of decoding tables. To that end we assume that we

have a certain state σ such that for every $c \in C$ and $\alpha \in \mathbf{O}_\perp^c$

$$elt(\beta, [\alpha]_c) = \begin{cases} \alpha & \text{if } \alpha \in \sigma^{(c)} \\ \perp & \text{otherwise.} \end{cases}$$

where $\beta \in \mathbf{O}^{c^*}$ is the value of bij^c in a certain ω . So every existing object of class c occurs in the sequence denoted by bij^c at a position which equals its code number. It is important to note that we cannot express this property of the sequence denoted by bij^c in the assertion language: There exists no assertion $P(bij^c)$ such that for every σ, δ, ω with $OK(\sigma, \delta, \omega)$ we have $\sigma, \delta, \omega \models P(bij^c)$ precisely if the above property holds. This is because at the level of the assertion language objects simply are not integers. Fortunately we shall not need the expressibility of exactly this property, but only of this property modulo an *osi*. This is the subject of section B.3.

Definition B.2

Let z^α, z^σ be some logical integer variables. We assume that the value of z^σ equals the code $[\sigma]$ of some state σ , and that the value of z^α equals $[\alpha]_c$ for some $\alpha \in \sigma^{(c)}$. For every logical expression l_d^c we define $l_d^c[z^\alpha, z^\sigma]$ as a triple $(\bar{i}, l_1^c_{\text{Bool}}, l_2^c_{\text{Int}})$, where \bar{i} denotes a sequence of logical integer variables, and l_1 and l_2 are logical expressions. Note that we do *not* define this transformation for logical expressions of type d^* with $d \in C^+$. The idea behind this transformation $l_d^c[z^\alpha, z^\sigma] = (\bar{i}, l_1, l_2)$ can be described as follows: The expression l_1 is constructed such that it is only true if the variables \bar{i} contain the code numbers of certain objects that are relevant for the evaluation of l_d^c . To do this, l_1 can consult the variables \overline{bij} as a translation table from code numbers to actual objects. Using this information, l_2 is a translation of l_d^c such that every operation on objects described by l_d^c is translated into a corresponding *arithmetical* operation on code numbers.

Here is the formal definition:

- $x_d^c[z^\alpha, z^\sigma] = (\epsilon, \text{true}, Val_d^c(z^\alpha, k, z^\sigma))$,
where $k = [x_d^c]$ and ϵ is the empty sequence.
- $u_d[z^\alpha, z^\sigma] = (\epsilon, \text{true}, Val_d(k, z^\sigma))$,
where $k = [u_d]$.
- $\text{nil}[z^\alpha, z^\sigma] = (\epsilon, \text{true}, 0)$.
- $\text{self}[z^\alpha, z^\sigma] = (\epsilon, \text{true}, z^\alpha)$.
- $l[z^\alpha, z^\sigma] = (\epsilon, \text{true}, Bc(l))$,
where $l = \text{true}, \text{false}$.
- $n[z^\alpha, z^\sigma] = (\epsilon, \text{true}, [n]_{\text{Int}})$.

- $z_{\text{Int}}[z^\alpha, z^\sigma] = (\epsilon, \text{true}, Ic(z_{\text{Int}}))$,
 $z_{\text{Bool}}[z^\alpha, z^\sigma] = (\epsilon, \text{true}, Bc(z_{\text{Bool}}))$.
- $z_c[z^\alpha, z^\sigma] = (\langle i \rangle, \text{if } z_c \doteq \text{nil then } i \doteq 0 \text{ else } \text{bij}^c \cdot i = z_c \text{ fi}, i)$.
- $(l_c' \cdot x_d')[z^\alpha, z^\sigma] = (\bar{i}, l_1, Val_d'(l_2, k, z^\sigma))$,
 where $k = [x_d']$ and $l_c'[z^\alpha, z^\sigma] = (\bar{i}, l_1, l_2)$.
- $(l_{\text{Int}}^* \cdot l_{\text{Int}})[z^\alpha, z^\sigma] = (\bar{i}, l_1, Ic(l_{\text{Int}}^* \cdot Id(l_2)))$,
 $(l_{\text{Bool}}^* \cdot l_{\text{Int}})[z^\alpha, z^\sigma] = (\bar{i}, l_1, Bc(l_{\text{Bool}}^* \cdot Id(l_2)))$,
 where $l_{\text{Int}}[z^\alpha, z^\sigma] = (\bar{i}, l_1, l_2)$.
- $(l_d^* \cdot l_{\text{Int}})[z^\alpha, z^\sigma] = (\bar{i} \circ \langle j \rangle, l_1 \wedge \text{if } l_d^* \cdot Id(l_2) \doteq \text{nil then } j \doteq 0 \text{ else } \text{bij}^d \cdot j \doteq l_d^* \cdot Id(l_2) \text{ fi}, j)$,
 where $d \in C$, $l_{\text{Int}}[z^\alpha, z^\sigma] = (\bar{i}, l_1, l_2)$ and j is a fresh integer logical variable.
- $(l_1 + l_2)[z^\alpha, z^\sigma] = (\bar{i}, l_1 \wedge l_2', Ic(Id(l_1) + Id(l_2')))$
 where $l_1[z^\alpha, z^\sigma] = (\bar{i}_1, l_{11}, l_{12})$, $l_2[z^\alpha, z^\sigma] = (\bar{i}_2, l_{21}, l_{22})$, $\bar{i} = \bar{i}_1 \circ \bar{j}$, \bar{j} is some sequence of new logical integer variables of the same length as \bar{i}_2 , $l_2' = l_{21}[\bar{j}/\bar{i}_2]$, and $l_2'' = l_{22}[\bar{j}/\bar{i}_2]$.
- if l_1 then l_2 else l_3 fi $[z^\alpha, z^\sigma] = (\bar{i}, l_1 \wedge l_2' \wedge l_3'$, if l_1 then l_2' else l_3' fi)
 where $l_1[z^\alpha, z^\sigma] = (\bar{i}_1, l_{11}, l_{12})$, $l_2[z^\alpha, z^\sigma] = (\bar{i}_2, l_{21}, l_{22})$, $l_3[z^\alpha, z^\sigma] = (\bar{i}_3, l_{31}, l_{32})$,
 $\bar{i} = \bar{i}_1 \circ \bar{j}_2 \circ \bar{j}_3$, \bar{j}_2 and \bar{j}_3 are sequences of new logical variables of the same length as \bar{i}_2 , \bar{i}_3 , respectively, such that \bar{i}_1 , \bar{j}_2 and \bar{j}_3 are mutually disjoint,
 $l_2' = l_{21}[\bar{j}_2/\bar{i}_2]$, $l_2'' = l_{22}[\bar{j}_2/\bar{i}_2]$, $l_3' = l_{31}[\bar{j}_3/\bar{i}_3]$, and $l_3'' = l_{32}[\bar{j}_3/\bar{i}_3]$.
- $(l_1 \doteq l_2)[z^\alpha, z^\sigma] = (\bar{i}, l_1 \wedge l_2', l_{12} \doteq l_{22}')$
 where $l_1[z^\alpha, z^\sigma] = (\bar{i}_1, l_{11}, l_{12})$, $l_2[z^\alpha, z^\sigma] = (\bar{i}_2, l_{21}, l_{22})$, $\bar{i} = \bar{i}_1 \circ \bar{j}$, \bar{j} is some sequence of new logical integer variables of the same length as \bar{i}_2 , $l_2' = l_{21}[\bar{j}/\bar{i}_2]$, and $l_2'' = l_{22}[\bar{j}/\bar{i}_2]$.

Next we define for every assertion P^c its transformation $P^c[z^\alpha, z^\sigma]$.

- $l_{\text{Bool}}^c[z^\alpha, z^\sigma] = \exists \bar{i}(l_1 \wedge l_2 \doteq Bc(\text{true}))$,
 where $l_{\text{Bool}}^c[z^\alpha, z^\sigma] = (\bar{i}, l_1, l_2)$.
- $(P_1 \wedge P_2)[z^\alpha, z^\sigma] = P_1[z^\alpha, z^\sigma] \wedge P_2[z^\alpha, z^\sigma]$.
- $(\exists z_a P)[z^\alpha, z^\sigma] = \exists z_a P[z^\alpha, z^\sigma]$
 for $a = \text{Int}, \text{Bool}, \text{Int}^*, \text{Bool}^*$.
- $(\exists z_d P)[z^\alpha, z^\sigma] = \exists z_d(z_d \in \text{bij}^d \wedge P[z^\alpha, z^\sigma])$
 for every $d \in C$. Here $z_d \in \text{bij}^d$ abbreviates $\exists i z_d \doteq \text{bij}^d \cdot i$ (cf. definition 6.28).
- $(\exists z_d^* P)[z^\alpha, z^\sigma] = \exists z_d^*(z_d^* \subseteq \text{bij}^d \wedge P[z^\alpha, z^\sigma])$
 for every $d \in C$. Here $z_d^* \subseteq \text{bij}^d$ abbreviates $\forall i z_d^* \cdot i \in \text{bij}^d$ (cf. definition 6.28).

- $(\forall z_a P)[z^\alpha, z^\sigma] = \forall z_a P[z^\alpha, z^\sigma]$
for $a = \text{Int}, \text{Bool}, \text{Int}^*, \text{Bool}^*$.
- $(\forall z_d P)[z^\alpha, z^\sigma] = \forall z_d (z_d \in \text{bij}^d \rightarrow P[z^\alpha, z^\sigma])$
for every $d \in C$.
- $(\forall z_{d^*} P)[z^\alpha, z^\sigma] = \forall z_{d^*} (z_{d^*} \subseteq \text{bij}^d \rightarrow P[z^\alpha, z^\sigma])$
for every $d \in C$.

In this transformation we assume that the quantified variables are distinct from any of the variables of $\overline{\text{bij}}$. Note that the result of this transformation applied to an arbitrary assertion is a quantification-restricted assertion.

To describe the semantics of this transformation we need the following definition.

Definition B.3

Let $\omega \in \Omega, \sigma \in \Sigma$, and let f be an *osi* (see definition 6.17). Then we write $\text{Code}(\omega, \sigma, f)$ iff for every $c \in C$ we have

- $\sigma_\perp^{(c)} = \{\text{elt}(\beta^{c^*}, n) : n \in \mathbb{N}\}$
- for all $\alpha \in \sigma^{(c)}$ and for all $n \in \mathbb{N}$ we have

$$\text{elt}(\beta^{c^*}, n) = \alpha \quad \text{iff} \quad f^c(\alpha) = n$$

where $\beta^{c^*} = \omega(\text{bij}^c)$.

We write $\text{Code}_L(\omega, \sigma, f)$ if $\text{Code}(\omega, \sigma, f)$ and additionally for every $c \in C$ we have

- $\omega(z) \in \omega(\text{bij}^c)$ for every $z \in L \cap LVar_c$
- $\omega(z) \subseteq \omega(\text{bij}^c)$ for every $z \in L \cap LVar_{c^*}$.

In a sense $\text{Code}(\omega, \sigma, f)$ can be interpreted as saying that $\omega(\text{bij}^c)$ codes the restriction of the *osi* f to the existing objects of σ .

Now we are ready for the following semantical interpretation of the transformation described above.

Theorem B.4

Assume to be given the states $\sigma, \sigma', \sigma''$ such that $\sigma \preceq \sigma''$ and an *osi* f such that $f(\sigma^{(c)}) = \sigma'^{(c)}$ for every $c \in C$. Furthermore let $\omega \in \Omega$ and $\delta \in \Delta^c$ be such that

$OK(\omega, \delta, \sigma'')$ and $Code_L(\omega, \sigma, f)$, where $BVar \subseteq L \subseteq LVar$. Then for every assertion P^c such that $LVar(P^c) \subseteq L$ and $LVar(P^c) \cap \overline{bij} = \emptyset$ we have

$$\sigma', \delta', \omega' \models P^c \quad \text{iff} \quad \sigma'', \delta, \omega\{n, m/z^\alpha, z^\sigma\} \models P^c[z^\alpha, z^\sigma],$$

where $\delta' = f(\delta)$, $\omega' = f(\omega) \downarrow L$, $n = [f(\delta_{(1)})]_c$, $m = [\sigma']$, and z^α, z^σ are new logical integer variables.

Proof

Induction on the complexity of P^c . The case $P^c = l_{\text{Bool}}^c$ is treated as follows. For every logical expression l_d^c such that $LVar(l_d^c) \subseteq L$ and $LVar(l_d^c) \cap \overline{bij} = \emptyset$ we prove, by induction on the complexity of l_d^c , the following: Let $l_d^c[z^\alpha, z^\sigma] = (\bar{i}, l_1, l_2)$ where $\bar{i} = i_1, \dots, i_q$. Then there exists a unique sequence of natural numbers $\bar{k} = k_1, \dots, k_q$ such that

$$\mathcal{L}[l_1](\omega\{\bar{k}, n, m/\bar{i}, z^\alpha, z^\sigma\})(\delta)(\sigma'') = t$$

and for this \bar{k} we have

$$[\mathcal{L}[l_d^c](\omega')(\delta')(\sigma')]_d = \mathcal{L}[l_2](\omega\{\bar{k}, n, m/\bar{i}, z^\alpha, z^\sigma\})(\delta)(\sigma'').$$

□

B.3 Expressing the coding relationship

In this section we show how to express in the assertion language the relationship between a state and its code number. In definition B.6 we shall define the assertion $Bij(z^\sigma)$, which expresses, as accurately as possible, that the current state is coded by the value of z^σ and that the logical variables \overline{bij} form a correct decoding table. However, it is only possible to express this up to isomorphism, as we shall see in lemma B.7.

Definition B.5

First we define the following auxiliary assertions:

- $CI_{\text{Int}}^c(x_{\text{Int}}^c, z^\alpha, z^\sigma) = Ic((bij^c \cdot z^\alpha) \cdot x_{\text{Int}}^c) \doteq Val_{\text{Int}}^c(z^\alpha, k, z^\sigma)$,
 $CI_{\text{Bool}}^c(x_{\text{Bool}}^c, z^\alpha, z^\sigma) = Ic((bij^c \cdot z^\alpha) \cdot x_{\text{Bool}}^c) \doteq Val_{\text{Bool}}^c(z^\alpha, k, z^\sigma)$,
 where $k = [x]$.
- $CI_d^c(x_d^c, z^\alpha, z^\sigma) =$
 $((bij^c \cdot z^\alpha) \cdot x_d^c \doteq \text{nil} \rightarrow Val_d^c(z^\alpha, k, z^\sigma) \doteq 0) \wedge$
 $((bij^c \cdot z^\alpha) \cdot x_d^c \neq \text{nil} \rightarrow \forall p((bij^c \cdot z^\alpha) \cdot x_d^c \doteq bij^d \cdot p \rightarrow Val_d^c(z^\alpha, k, z^\sigma) \doteq p)),$
 where $d \in C$ and $k = [x_d^c]$
- $CT_{\text{Int}}(u_{\text{Int}}, z^\sigma) = Ic(u_{\text{Int}}) \doteq Val_{\text{Int}}(k, z^\sigma)$,
 $CT_{\text{Bool}}(u_{\text{Bool}}, z^\sigma) = Ic(u_{\text{Bool}}) \doteq Val_{\text{Bool}}(k, z^\sigma)$,
 where $k = [u]$.

- $CT_d(u_d, z^\sigma) =$
 $(u_d \doteq \text{nil} \rightarrow Val_d(k, z^\sigma) \doteq 0) \wedge$
 $(u_d \not\doteq \text{nil} \rightarrow \forall p(bij^d \cdot p \doteq u_d \rightarrow Val_d(k, z^\sigma) = p)),$
 where $d \in C$ and $k = [u_d]$

Definition B.6

Next we define the assertion $Bij(z^\sigma)$, where z^σ is some logical integer variable, as follows.

$$\begin{aligned}
 Bij(z^\sigma) = & \bigwedge_c \forall z_c \exists ! i (bij^c \cdot i \doteq z_c) \wedge \\
 & \bigwedge_c \forall i (E^c(i, z^\sigma) \leftrightarrow bij^c \cdot i \neq \text{nil}) \wedge \\
 & \bigwedge_c \forall i (bij^c \cdot i \neq \text{nil} \rightarrow \bigwedge_d \bigwedge_{x \in IVar_d^c} CI_d^c(x, i, z^\sigma)) \wedge \\
 & \bigwedge_d \bigwedge_{u \in TVar_d} CT_d(u, z^\sigma)
 \end{aligned}$$

The first conjunct states that for every c the sequence denoted by bij^c stores each existing object of class c exactly once. The second conjunct then can be interpreted as stating that every existing object of class c occurs in the sequence denoted by bij^c at a position which equals the code of *some* object that exists in the state coded by z^σ . The third conjunct relates the local state of every existing object with the one of its corresponding code. Finally, the fourth conjunct relates the values of the temporary variables with their coded versions.

In the following lemma we show how this assertion $Bij(z)$ can be used to describe the isomorphism between two states.

Lemma B.7

Let σ, ω, f such that $OK(\omega, \sigma)$, $Code(\omega, \sigma, f)$ and $\omega(z) = [\sigma']$.

Then:

$$\sigma, \delta, \omega \models Bij(z) \text{ iff } f(\sigma) = \sigma',$$

for an arbitrary δ such that $OK(\sigma, \delta, \omega)$.

Proof

Straightforward. □

B.4 Expressing the strongest postcondition

Finally we are ready for the theorem stating the expressibility of the strongest postcondition.

Theorem B.8

Let ρ^c be closed, $BVar \subseteq L \subseteq LVar$, P^c such that $LVar(P^c) \subseteq L$ and $\overline{bij} \cap L = \emptyset$.
 Then: $SP_L^c(\rho^c, P^c) = \exists bij^{c_1}, \dots, bij^{c_n}, z_1, z_2, z_3(Q)$ (assuming $C = \{c_1, \dots, c_n\}$),
 where $Q = \bigwedge_{1 \leq p \leq 5} Q_p$, and

- $Q_1 = T([\rho^c], z_1, z_2, z_3)$,
- $Q_2 = Bij(z_3)$,
- $Q_3 = bij^c \cdot A^c(z_1) \doteq \text{self}$,
- $Q_4 = \bigwedge_c \forall i (B^c(i, z_1) \leftrightarrow bij^c \cdot i \in b_c)$,
- $Q_5 = \exists z_{c_1}^*, \dots, z_{c_n}^* \bigwedge_{1 \leq p \leq 4} R_p$,
 where
 - $R_1 = \bigwedge_c (z_{c^*} \subseteq bij^c)$
 - $R_2 = \bigwedge_c \forall i (E^c(i, z_2) \leftrightarrow z_{c^*} \cdot i \neq \text{nil})$
 - $R_3 = \bigwedge_c (\bigwedge_{z'_c \in L} (z'_c \in z_{c^*}) \wedge \bigwedge_{z'_c \in L} (z'_c \subseteq z_{c^*}))$
 - $R_4 = P^c[z, z'] [\bar{z}/\overline{bij}, A^c(z_1)/z, z_2/z']$

where $l_{1a} \subseteq l_{2a}$, for $a = d^*$, abbreviates the assertion $\forall i (l_{1a} \cdot i \doteq \text{nil} \vee l_{1a} \cdot i \doteq l_{2a} \cdot i)$,
 and \bar{z} denotes a sequence $z_{c_1}^*, \dots, z_{c_n}^*$ of fresh logical variables.

The quantification $\exists bij^{c_1}, \dots, bij^{c_n}$ will correspond to the phrase (in theorem 6.21) “there exists an *osi* f ”. The variables z_1, z_2, z_3 will correspond to δ', σ_0 , and $f(\sigma)$, respectively. The conjunction $\bigwedge_{1 \leq i \leq 4} Q_i$ then expresses $f(\sigma) = \mathcal{P}[\rho](\gamma)(\delta')(\sigma_0)$. Finally, the assertion Q_5 expresses $\sigma_0, \delta', \omega' \models P$, where $\omega' = f(\omega) \upharpoonright L$. Let us look into this more closely. The conjunction $R_1 \wedge R_2$ states that the variable z_{c_i} , $1 \leq i \leq n$, stores all the existing objects of σ_0 (of class c_i) at a position which equals its code. The assertion R_3 then states that ω' is compatible with σ_0 . Finally, the assertion R_4 expresses that $\sigma_0, \delta', \omega' \models P$.

Proof

Let $\sigma, \delta, \omega \models SP_L^c(\rho^c, P^c)$. So there exists for $i = 1, \dots, n$, $\alpha_i \in \mathbf{O}^{c_i}$, and $\beta_1, \beta_2, \beta_3 \in \mathbf{N}$ such that $\sigma, \delta, \omega' \models Q$, where $\omega' = \omega \{ \alpha_i / bij^{c_i} \}_i \{ \beta_1, \beta_2, \beta_3 / z_1, z_2, z_3 \}$.

As $\sigma, \delta, \omega' \models Q_1$ there exists $\sigma_0, \sigma_1, \delta'$ such that $\sigma_1 = \mathcal{P}[\rho^c](\gamma)(\delta')(\sigma_0)$, γ arbitrary, and $[\delta'] = \beta_1$, $[\sigma_0] = \beta_2$, $[\sigma_1] = \beta_3$.

Now let f be an *osi* such that for $\alpha \in \sigma^{(c_i)}$ we have: $f(\alpha) = \beta$ iff $\text{elt}(\omega'(bij^{c_i}), \beta) = \alpha$. (Note that as $\sigma, \delta, \omega' \models Q_2$ we have that for $\alpha \in \sigma^{(c_i)}$ there exists some $\beta \in \mathbf{N}$ such that $\text{elt}(\alpha_i, \beta) = \alpha$, furthermore we have $E^{c_i}(\beta, \beta_3)$ so $\beta \in \sigma_1^{(c_i)}$.) So we have $\text{Code}(\omega', \sigma, f)$ and by lemma B.7 we infer $f(\sigma) = \sigma_1$.

From $\sigma, \delta, \omega' \models Q_3$ it follows that $\delta'_{(1)} = f(\delta_{(1)})$. Furthermore from $\sigma, \delta, \omega' \models Q_4$ it follows that $\delta'_{(2)(c)} = \{f^c(\alpha) : \alpha \in \omega(b_c)\}$. Note that $OK(\omega, \delta, \sigma)$ so we infer that $\delta' = f(\delta)$.

Finally, we have $\sigma, \delta, \omega' \models Q_5$. So there exists for $i = 1, \dots, n$, $\alpha'_i \in \mathbf{O}^{c_i}$, $\omega'' = \omega'\{\alpha'_i/z_{c_i}\}_i$ such that $\sigma, \delta, \omega'' \models \bigwedge_{1 \leq j \leq 4} R_j$. Let σ' such that, for an arbitrary c , $\sigma'^{(c)} = f^{-1c}(\sigma_0^{(c)})$. It then follows that $\sigma' \preceq \sigma$ and by $\sigma, \delta, \omega'' \models \bigwedge_{1 \leq j \leq 3} R_j$ we have $Code_L(\omega'', \sigma', f)$, where $\omega''' = \omega\{\alpha'_i/bij^{c_i}\}_i\{\delta'_{(1)}, \beta_2/z, z'\}$. Furthermore we have $\sigma, \delta, \omega''' \models P^c[z, z']$, so we have by theorem B.4: $\sigma_0, \delta', \bar{\omega} \models P^c$, where $\bar{\omega} = f(\omega''') \downarrow L = f(\omega) \downarrow L$. This finishes one part of the proof.

On the other hand, let $\sigma, \sigma_0, \delta, \omega, f$ such that:

- $f(\sigma) = \mathcal{P}[\rho^c](\gamma)(\delta')(\sigma_0)$, γ arbitrary.
- $\sigma_0, \delta', \omega' \models P^c$.

where $\delta' = f(\delta)$ and $\omega' = f(\omega) \downarrow L$.

Let $\beta_1 = [\delta']$, $\beta_2 = [\sigma_0]$, $\beta_3 = [f(\sigma)]$ and $\alpha_i \in \mathbf{O}^{c_i}$, for $i = 1, \dots, n$ (assuming $C = \{c_1, \dots, c_n\}$), such that $elt(\alpha_i, m) = \alpha (\neq \perp)$ iff $\alpha \in \sigma^{(c_i)}$ and $f^{c_i}(\alpha) = m$. Furthermore let $\omega'' = \omega\{\alpha_i/bij^{c_i}\}_i\{\beta_1, \beta_2, \beta_3/z_1, z_2, z_3\}$.

Now $f(\sigma) = \mathcal{P}[\rho^c](\gamma)(\delta')(\sigma_0)$ so we have $\sigma, \delta, \omega'' \models Q_1$.

We have $Code(\omega'', \sigma, f)$, and $\omega''(z_3) = [f(\sigma)]$, so by lemma B.7 we have $\sigma, \delta, \omega'' \models Q_2$.

From $\delta' = f(\delta)$, $OK(\sigma, \delta, \omega)$ and $OK(\sigma_0, \delta')$ it easily follows that $\sigma, \delta, \omega'' \models Q_3 \wedge Q_4$.

Let, for $i = 1, \dots, n$, α'_i be a subsequence of α_i , such that $\sigma_0^{(c_i)} = \{\alpha : elt(\alpha'_i, \alpha) \neq \perp\}$. Furthermore let $\omega''' = \omega''\{\alpha'_i/z_{c_i}\}_i$. Now from α'_i being a subsequence of α_i it immediately follows that $\sigma, \delta, \omega''' \models R_1$.

From $\sigma_0^{(c_i)} = \{\alpha \in \mathbf{O}^{c_i} : elt(\alpha'_i, \alpha) \neq \perp\}$ it in turn follows that $\sigma, \delta, \omega''' \models R_2$. Furthermore we have that σ_0 and ω' are compatible, and $\omega' = f(\omega) \downarrow L = f(\omega''') \downarrow L$, from which it follows that: $\sigma, \delta, \omega''' \models R_3$.

Finally, let σ' be such that for an arbitrary c we have $\sigma'^{(c)} = f^{-1c}(\sigma_0^{(c)})$ and $\bar{\omega} = \omega'''\{\alpha'_i/bij^{c_i}\}_i\{\delta'_{(1)}, \beta_2/z, z'\}$. We then have that $Code_L(\bar{\omega}, \sigma', f)$ and $\sigma' \preceq \sigma$. So from $\sigma_0, \delta', \omega' \models P^c$ and $\omega' = f(\bar{\omega}) \downarrow L$ applying theorem B.4 it follows that $\sigma, \delta, \bar{\omega} \models P^c[z, z']$. So we infer that $\sigma, \delta, \omega''' \models R_4$.

Summerizing we conclude that: $\sigma, \delta, \omega \models SP_L^c(\rho^c, P^c)$. □

C A closure property of the semantics

In this appendix we prove a closure property of the semantics with respect to object-space isomorphisms. To get started it turns out to be convenient to have the following definition.

Definition C.1

Let $\beta_1^{d_1}, \dots, \beta_n^{d_n}$ be some sequence of objects. We define $OK(\beta_1^{d_1}, \dots, \beta_n^{d_n}, \delta, \sigma)$ iff $OK(\delta, \sigma)$ and additionally $\beta_i \in \sigma^{(d_i)}$, $i = 1, \dots, n$.

Definition C.2

For

- $\mathcal{F} \in \left(\prod_{i=1}^n \mathbf{O}_{\perp}^{d_i} \right) \rightarrow \Delta^c \rightarrow \Sigma_{\perp} \rightarrow \left(\Sigma_{\perp} \times \mathbf{O}_{\perp}^{d_0} \right)$, for some c, n, d_0, \dots, d_n ,
- $\mathcal{G} \in \Delta^c \rightarrow \Sigma_{\perp} \rightarrow \left(\Sigma_{\perp} \times \mathbf{O}_{\perp}^d \right)$, for some c, d ,
- $\mathcal{H} \in \Delta^c \rightarrow \Sigma_{\perp} \rightarrow \Sigma_{\perp}$, for some c ,

we define

- $Cl(\mathcal{F})$ iff for an arbitrary $\beta_0^{d_0}, \dots, \beta_n^{d_n}, \delta, \sigma, \sigma', f$ such that $OK(\beta_1, \dots, \beta_n, \delta, \sigma)$:
if $\mathcal{F}(\beta_1, \dots, \beta_n)(\delta)(\sigma) = \langle \sigma', \beta_0 \rangle$
then there exists an *osi* g such that $f^c \downarrow \sigma^{(c)} = g^c \downarrow \sigma^{(c)}$, for an arbitrary c , and
 $\mathcal{F}(f^{d_1}(\beta_1), \dots, f^{d_n}(\beta_n))(f(\delta))(f(\sigma)) = \langle g(\sigma'), g^{d_0}(\beta_0) \rangle$,
- $Cl(\mathcal{G})$ iff for an arbitrary $\beta, \delta, \sigma, \sigma', f$ such that $OK(\delta, \sigma)$:
if $\mathcal{G}(\delta)(\sigma) = \langle \sigma', \beta \rangle$
then there exists an *osi* g such that $f^c \downarrow \sigma^{(c)} = g^c \downarrow \sigma^{(c)}$, for an arbitrary c , and
 $\mathcal{G}(f(\delta))(f(\sigma)) = \langle g(\sigma'), g^{d_0}(\beta) \rangle$,
- $Cl(\mathcal{H})$ iff for an arbitrary $\delta, \sigma, \sigma', f$ such that $OK(\delta, \sigma)$:
if $\mathcal{H}(\delta)(\sigma) = \sigma'$
then there exists an *osi* g such that $f^c \downarrow \sigma^{(c)} = g^c \downarrow \sigma^{(c)}$, for an arbitrary c , and
 $\mathcal{H}(f(\delta))(f(\sigma)) = g(\sigma')$.

(Here \downarrow denotes the restriction operator.)

Now we are ready to analyse this closure property denoted by Cl . We start with the following lemma which states that the meaning of an arbitrary expression $s \in SExp$ satisfies this property assuming it holds for the meaning assigned to an arbitrary method:

Lemma C.3

Let γ be an environment such that for an arbitrary method name m we have $Cl(\gamma(m))$. Then for every expression $s \in SExp$ we have $Cl(\mathcal{Z}[s](\gamma))$.

Proof

The proof proceeds by induction on the complexity of s :

$s = e$: Note that we have by theorem 6.21 $\mathcal{E}[e](\delta)(\sigma) = \mathcal{E}[e](f(\delta))(f(\sigma))$ for an arbitrary δ, σ such that $OK(\delta, \sigma)$.

$s = \text{new}_d$: Let $\mathcal{Z}[\text{new}_d](\gamma)(\delta)(\sigma) = \langle \sigma', \beta \rangle$. So we have $\text{pick}^d(\sigma^{(d)}) = \beta$. Let $\beta' = \text{pick}^{(d)}(f(\sigma)^{(d)})$ and g be an *osi* such that $f^c \downarrow \sigma^{(c)} = g^c \downarrow \sigma^{(c)}$, for an arbitrary c , and $g^d(\beta) = \beta'$. It follows that $\mathcal{Z}[\text{new}_d](\gamma)(f(\delta))(f(\sigma)) = \langle g(\sigma'), \beta' \rangle$.

$s = e_0!m(e_1, \dots, e_n)$: Let for $i = 0, \dots, n$ $\mathcal{E}[e_i](\delta)(\sigma) = \beta_i$ ($OK(\delta, \sigma)$) and $\gamma(m)(\beta_1, \dots, \beta_n)(\delta')(\sigma) = \langle \sigma', \beta \rangle$, where

$$\begin{aligned} \delta'_{(1)} &= \beta_0 \\ \delta'_{(2)(c')} &= \delta_{(2)(c')} \{ \delta_{(2)(c')} \cup \delta_{(1)}/c' \} & c' = c \\ \delta'_{(2)(c')} &= \delta_{(2)(c')} & c' \neq c, \end{aligned}$$

assuming $s \in SExp_d^c$, for some d .

As we have $Cl(\gamma(m))$ it follows that $\gamma(m)(f(\beta_1), \dots, f(\beta_n))(f(\delta'))(f(\sigma)) = \langle g(\sigma'), g(\beta) \rangle$, for some *osi* g such that $g^c \downarrow \sigma^{(c)} = f^c \downarrow \sigma^{(c)}$, c arbitrary. (Note that by lemma 3.21 and $OK(\delta, \sigma)$ we have $OK(\beta_1, \dots, \beta_n, \delta', \sigma)$.) By theorem 6.21 we have $\mathcal{E}[e_i](f(\delta))(f(\sigma)) = f(\beta_i)$. Furthermore we have

$$\begin{aligned} f(\delta')_{(1)} &= f(\beta_0) \\ f(\delta')_{(2)(c')} &= f^{c'}(\delta_{(2)(c')}) \{ f^{c'}(\delta_{(2)(c')}) \cup f^{c'}(\delta_{(1)})/c' \} & c' = c \\ f(\delta')_{(2)(c')} &= f^{c'}(\delta_{(2)(c')}) & c \neq c'. \end{aligned}$$

So we conclude $\mathcal{E}[s](\gamma)(f(\delta))(f(\sigma)) = \langle g(\sigma'), g(\beta) \rangle$. □

Next we prove the closure property Cl for the meaning assigned to statements assuming it holds for the one assigned to expressions.

Lemma C.4

Let γ be an agreement-preserving environment such that for an arbitrary $s \in SExp$ we have $Cl(\mathcal{Z}[s](\gamma))$. Then we have $Cl(\mathcal{S}[S](\gamma))$ for an arbitrary $S \in Stat$.

Proof

The proof proceeds by induction on the complexity of S . We treat the following cases:

$S = x_d^c \leftarrow s_d^c$: Let $\mathcal{S}[S](\gamma)(\delta)(\sigma) = \sigma'' (OK(\delta, \sigma))$ and f be some *osi*. So we have $\mathcal{Z}[s](\gamma)(\delta)(\sigma) = \langle \sigma', \beta \rangle$ such that $\sigma'' = \sigma' \{ \beta / \delta_{(1)}, x \}$. By $Cl(\mathcal{Z}[s](\gamma))$ it then follows that there exists an *osi* g such that $g^c \downarrow \sigma^{(c)} = f^c \downarrow \sigma^{(c)}$, for an arbitrary c , and $\mathcal{Z}[s](\gamma)(f(\delta))(f(\sigma)) = \langle g(\sigma'), g(\beta) \rangle$. Now $g(\sigma'') = g(\sigma') \{ g^d(\beta) / g^c(\delta_{(1)}, x \}$, so we conclude $\mathcal{S}[S](\gamma)(f(\delta))(f(\sigma)) = g(\sigma'')$.

$S = S_1; S_2$: Let $\mathcal{S}[S](\gamma)(\delta)(\sigma) = \sigma' (OK(\delta, \sigma))$ and f be some *osi*. So there exists a σ'' such that $\mathcal{S}[S_1](\gamma)(\delta)(\sigma) = \sigma''$ and $\mathcal{S}[S_2](\gamma)(\delta)(\sigma'') = \sigma'$. By the induction hypothesis we have for some *osi* g such that $g^c \downarrow \sigma^{(c)} = f^c \downarrow \sigma^{(c)}$ for an arbitrary c and $\mathcal{S}[S_1](\gamma)(f(\delta))(f(\sigma)) = g(\sigma'')$. Another application of the induction hypothesis gives us an *osi* h such that $h^c \downarrow \sigma''^{(c)} = g^c \downarrow \sigma''^{(c)}$, for an arbitrary c , and $\mathcal{S}[S_2](\gamma)(g(\delta))(g(\sigma'')) = h(\sigma')$. Putting these applications of the induction hypothesis together gives us $h^c \downarrow \sigma^{(c)} = f^c \downarrow \sigma^{(c)}$, for an arbitrary c , and $\mathcal{S}[S](\gamma)(f(\delta))(f(\sigma)) = h(\sigma')$. (Note that by lemma 3.21 $\sigma \preceq \sigma''$ and, as $OK(\delta, \sigma)$, $f(\delta) = g(\delta)$.)

$S = \text{while } e \text{ do } S_1 \text{ od}$: Let $\mathcal{S}[S](\gamma)(\delta)(\sigma) = \sigma'$. So we have $\mu\Phi(\delta)(\sigma) = \sigma'$, where Φ is as defined in definition 3.14. Now it suffices to prove that for an arbitrary $\varphi \in \Delta^c \rightarrow (\Sigma_{\perp} \rightarrow \Sigma_{\perp})$ such that $Cl(\varphi)$ we have $Cl(\Phi(\varphi))$. So assume for some φ we have $Cl(\varphi)$. Let $\Phi(\varphi)(\delta)(\sigma) = \sigma' (OK(\delta, \sigma))$ and f be some *osi*. We consider the case that $\mathcal{E}[e](\delta)(\sigma) = \mathbf{t}$. By theorem 6.21 we then have $\mathcal{E}[e](f(\delta))(f(\sigma)) = \mathbf{t}$. Furthermore we have $\varphi(\delta, \mathcal{S}[S_1](\gamma)(\delta)(\sigma)) = \sigma'$. Let $\mathcal{S}[S_1](\gamma)(\delta)(\sigma) = \sigma''$, by the induction hypothesis it then follows that for some *osi* g we have $g^c \downarrow \sigma^{(c)} = f^c \downarrow \sigma^{(c)}$, for an arbitrary c , and $\mathcal{S}[S_1](\gamma)(f(\delta))(f(\sigma)) = g(\sigma'')$. By assumption there exists also an *osi* h such that $h^c \downarrow \sigma''^{(c)} = g^c \downarrow \sigma''^{(c)}$, for an arbitrary c , and $\varphi(g(\delta), g(\sigma'')) = h(\sigma')$. Putting this together gives us $h^c \downarrow \sigma^{(c)} = f^c \downarrow \sigma^{(c)}$ for an arbitrary c and $\Phi(\varphi)(f(\delta))(f(\sigma)) = h(\sigma')$. (Note that by lemma 3.21 $\sigma \preceq \sigma''$ and, as $OK(\delta, \sigma)$, $f(\delta) = g(\delta)$.) \square

We proceed with the following lemma which states the closure property of the meaning of class definitions assuming it holds for the meaning of statements:

Lemma C.5

Let γ be an agreement-preserving environment such that $Cl(\mathcal{S}[S](\gamma))$ for an arbitrary statement S . Then we have for every method name m defined by D $Cl(\mathcal{C}[D](\gamma)(m))$ for an arbitrary class definition D .

Proof

Let $\gamma' = \mathcal{C}[D](\gamma)$ and f be some *osi*. Now let the method name m be defined by D , say m is declared as μ_{d_0, \dots, d_k}^c . We have $\gamma'(m) = \mathcal{M}[\mu_{d_0, \dots, d_k}^c](\gamma)$. Let $\mu_{d_0, \dots, d_k}^c = (u_1, \dots, u_k) : S \uparrow e$. Moreover let

$$\begin{aligned}
\mathcal{M}[(u_1, \dots, u_k) : S \uparrow e](\gamma)(\beta_1, \dots, \beta_k)(\delta)(\sigma) &= \langle \sigma''', \beta \rangle \\
\text{where } \sigma' &= \langle \sigma_{(1)}, \sigma_{(2)}, \sigma'_{(3)} \rangle \\
\sigma'_{(3)}(u) &= \beta_i && \text{if } u = u_i \\
&= \perp && \text{otherwise} \\
\sigma'' &= S^c[S](\gamma)(\delta)(\sigma') \\
\beta &= \mathcal{E}[e](\delta)(\sigma'') \\
\sigma''' &= \langle \sigma''_{(1)}, \sigma''_{(2)}, \sigma_{(3)} \rangle
\end{aligned}$$

Note that we assume $\sigma \neq \perp$ and $\delta_{(1)}$ not to be blocked. If one of these do hold we have $\sigma' = \perp$ from which follows that $\sigma''', \beta = \perp$. By the assumption about γ we have for some osi $g \downarrow \sigma^{(c)} = f^c \downarrow \sigma'^{(c)}$, for every arbitrary c , and $S^c[S](\gamma)(f(\delta))(f(\sigma')) = g(\sigma'')$. (Note that $OK(\beta_1, \dots, \beta_k, \delta, \sigma)$ implies $OK(\delta, \sigma')$.) As $\sigma'^{(c)} = \sigma^{(c)}$ for an arbitrary c we have $g^c(\sigma^{(c)}) = f^c(\sigma^{(c)})$ for an arbitrary c . By theorem 6.21 we have $g(\beta) = \mathcal{E}[e](g(\delta))(g(\sigma''))$. (Note that as γ is agreement-preserving we have by lemma 3.21 $\sigma \preceq \sigma''$, and so $OK(\delta, \sigma'')$.) Putting this together gives us

$$\mathcal{M}[(u_1, \dots, u_k) : S \uparrow e](\gamma)(f(\beta_1), \dots, f(\beta_k))(f(\delta))(f(\sigma)) = \langle g(\sigma'''), g(\beta) \rangle.$$

□

In the next lemma we prove that the meaning of units satisfies the closure property *Cl*.

Lemma C.6

Let $U = D_1, \dots, D_n$ be an unit such that every method occurring in it is defined by it. Then for every method name m we have $Cl(\gamma'(m))$, where $\mathcal{U}[U](\gamma_0) = \gamma'$ and γ_0 is the “empty” environment defined by

$$\gamma_0(\bar{\beta})(\delta)(\sigma) = \langle \perp, \perp \rangle.$$

Proof

We have $\gamma' = \sqcup_i \gamma_i$, γ_0 being the “empty” environment and $\mathcal{C}[D_1] \circ \dots \circ \mathcal{C}[D_n](\gamma_i) = \gamma_{i+1}$. We prove by induction that $Cl(\gamma_i(m))$, m arbitrary. From this it is not difficult to prove that $Cl(\gamma'(m))$.

$i = 0$: Evident.

$i = j + 1$: By the induction hypothesis we have $Cl(\gamma_j(m))$. Furthermore by lemma 3.21 we know that γ_i is agreement-preserving. From this follows by applying the lemmas C.3, C.4 and C.5 that $Cl(\gamma_{i+1}(m))$. (Note that lemma C.5 can be applied only for method names defined by U , but as we have $\gamma_i(m) = \gamma_0(m)$, for $i \in \mathbb{N}$ and m not defined by U this suffices.) □

We conclude this appendix with the following theorem which states the closure property of the meaning assigned to closed programs:

Theorem C.7

For an arbitrary closed program $\rho = \langle U | c : S \rangle$, environment γ we have $Cl(\mathcal{P}[\rho](\gamma))$.

Proof

First note that as ρ is a closed program we have $\mathcal{P}[\rho](\gamma) = \mathcal{P}[\rho](\gamma_0)$. We have by definition 3.18 that $\mathcal{P}[\rho](\gamma_0) = \mathcal{S}[S](\gamma')$, where $\gamma' = \mathcal{U}[U](\gamma_0)$. By lemma C.6 we have $Cl(\gamma'(m))$ for every method name m . So applying the lemmas C.3 and C.4 gives us $Cl(\mathcal{S}[S](\gamma'))$. (Note that by lemma 3.21 γ' is agreement-preserving.) \square

Corollary C.8

For an arbitrary closed program $\rho, \sigma, \sigma', \delta, f$ such that $\sigma' = \llbracket \rho \rrbracket(\gamma)(\delta)(\sigma)$ there exists an *osi* g such that $g^c \downarrow \sigma^{(c)} = f^c \downarrow \sigma^{(c)}$ and $g(\sigma') = \llbracket \rho \rrbracket(\gamma)(f(\delta))(f(\sigma))$.

A proof theory for process creation

Pierre America and Frank de Boer

Contents

1	Introduction	122
2	The programming language	125
3	The assertion language	130
3.1	The local assertion language	130
3.2	The global assertion language	132
3.3	Correctness formulas	134
4	The proof system	136
4.1	The local proof system	136
4.2	The intermediate proof system	137
4.3	The global proof system	144
5	An example proof	147
6	Semantics	151
6.1	Semantics of the assertion languages	151
6.2	The transition system	155
6.3	Truth of correctness formulas	159

7 Soundness	160
7.1 The intermediate proof system	160
7.2 The global proof system	164
8 Completeness	171
8.1 Histories	171
8.2 A most general proof outline	175
9 Expressibility	184
9.1 Coding techniques	184
9.2 Object-space isomorphisms	187
9.3 Coding the global invariant	189
9.4 Expressing preconditions and postconditions	194
10 Conclusion	197
References	197
A Index of notation	199
A.1 Sets and their typical elements	199
A.2 Syntactic transformations	200

1 Introduction

The goal of this paper is to develop a formal system for reasoning about the correctness of a certain class of parallel programs. We shall consider programs written in a programming language, which we simply call P. The language P is a simplified relative of POOL, a parallel object-oriented language [Am2]. POOL makes use of

the structuring mechanisms of object-oriented programming [Mey], integrated with concepts for expressing concurrency: processes and communication.

A program of our language P describes the behaviour of a whole system in terms of its constituents, *objects*. These objects have the following important properties: First of all, each object has an independent activity of its own: a local process that proceeds in parallel with all the other objects in the system. Second, new objects can be created at any point in the program. The identity of such a new object is at first only known to itself and its creator, but from there it can be passed on to other objects in the system. Note that this also means that the number of processes executing in parallel may increase during the evolution of the system.

Objects possess some internal data, which they store in *variables*. The value of a variable is either an element of a predefined data type (Int or Bool), or it is a *reference* to another object. The variables of one object are not accessible to other objects. The objects can interact only by sending *messages*. A message is transferred synchronously from the sender to the receiver. It contains exactly one value; this can be an integer or a boolean, or it can be a reference to an object. (This is the only essential difference between P and POOL: in POOL communication proceeds by a rendezvous mechanism, where a method, a kind of procedure, is invoked in the receiving object in response to a message.) Thus we see that a system described by a program in the language P consists of a dynamically evolving collection of objects, which are all executing in parallel, and which know each other by maintaining and passing around references. This means that the communication structure of the processes is determined dynamically, without any regular structure imposed on it a priori. This is in contrast to the static structure (a fixed number of processes, communicating with statically determined partners) in [AFR] and the tree-like structure in [ZREB].

One of the main proof theoretical problems of such an object-oriented language is how to reason about dynamically evolving *pointer structures*. We want to reason about these structures on an abstraction level that is *at least as high as that of the programming language*. In more detail, this means the following:

- The only operations on 'pointers' (references to objects) are
 - testing for equality
 - dereferencing (looking at the value of an instance variable of the referenced object)
- In a given state of the system, it is only possible to mention the objects that exist in that state. Objects that have not (yet) been created do not play a role.

Strictly speaking, direct dereferencing is not even allowed in the programming language, because each object has access to its own instance variables only. However, for

the time being we allow it in the assertion language. Otherwise, even more advanced techniques would be necessary to reason about the correctness of a program (see, e.g., [Am3]). Nevertheless, the proof system presented in this paper formalizes reasoning about dynamically evolving pointer structures at a more appropriate abstraction level than the system developed in [Bo] where those structures are described in terms of some particular coding of the objects.

The above restrictions have quite severe consequences for the proof system. The limited set of operations on pointers implies that first-order logic is too weak to express some interesting properties of pointer structures. Therefore we have to extend our assertion language to make it more expressive. We will do so by allowing the assertion language to reason about *finite sequences* of objects. (This is not uncommon in proof systems dealing with more data types than integers only [TZ].) Furthermore we have to define some special substitution operations to model aliasing and the creation of new objects.

To deal with parallelism, the proof theory we shall develop uses the concepts of *cooperation test*, *global invariant*, *bracketed section*, and *auxiliary variables*. These concepts have been developed in the proof theory of CSP [AFR], and have been applied to quite a variety of concurrent programming languages [HR]. Described very briefly, this proof method applied to our language consists of the following elements:

- A *local* stage. Here we deal with all statements that do not involve communication or object creation. These statements are proved correct with respect to pre- and postconditions in the usual manner of sequential programs [Ap1, Ba, Ho1]. At this stage, we use *assumptions* to describe the behaviour of the communication and creation statements. These will be verified in the next stage. In this local stage, a *local assertion language* is used, which only talks about the current object in isolation.
- An *intermediate* stage. In this stage the above assumptions about communication and creation statements are verified. Here a *global assertion language* is used, which reasons about all the objects in the system. For each creation statement and for each pair of possibly communicating send and receive statements it is verified that the specification used in the local proof system is consistent with the global behaviour.
- A *global* stage. Here some properties of the system as a whole can be derived from a kind of standard specification that arises from the intermediate stage. Again the global assertion language is used.

We have proved that the proof system is sound and complete with respect to a formally defined semantics. Soundness means that everything that can be proved using the proof system is indeed true in the semantics. On the other hand, completeness means

that every true property of a program that can be expressed using our assertion language can also be proved formally in the proof system. Due to the abstraction level of the assertion language we had to modify considerably the standard techniques for proving completeness.

Our paper is organized as follows: In the following section we describe the programming language P. In section 3 we define two assertion languages, the local one and the global one. Then, in section 4 we describe the proof system. Section 5 presents an example of a correctness proof for a nontrivial program. Section 6 presents the semantics of the programming language, of the assertion languages, and of the correctness formulas. In section 7 we prove the soundness of the proof system and in section 8 we prove completeness. The expressibility of the assertion languages is studied in section 9. Finally, in section 10 we draw some conclusions.

2 The programming language

In this section we define the programming language P of which we shall study the proof theory. This language is related to CSP [Ho2], in that it describes a number of processes that communicate synchronously by transmitting values to each other, but it has the additional possibility of dynamically creating processes and manipulating references to processes. It can also be compared to Smalltalk [GR], since it describes a dynamically evolving collection of objects where each has its own private data, but in P each object is provided with an autonomous local process and it communicates with other objects simply by exchanging a value instead of invoking a method.

A system, the result of executing a program written in P, consists of *objects*. On the one hand these objects have the properties of processes, that is, each of them has an internal activity, which runs in parallel with all the other objects in the system. On the other hand, objects are in some way like data records: they contain some internal data, and they have the ability to act on these data. An important characteristic of the objects in the language P is that they can be created dynamically: Whenever required during the execution of a program, a new object can be called into existence.

An object stores its internal data in *variables* (also called *instance variables* to distinguish them from the other kinds of variables that we shall need in the assertion language). A variable can contain a reference to an object, which can be another object, or possibly the object under consideration itself. Alternatively, it can contain an element of a standard, built-in data type, of which our language P contains only integers and booleans. The contents of a variable can be changed by an assignment statement. The variables of one object cannot be accessed directly by other objects. They can only be read and changed by the object to which they belong.

Interaction between objects takes place by sending messages. The language P uses a synchronous communication mechanism, that is, the sender and receiver of a message perform the communication at the same time; the one that reaches its communication statement first will wait for its partner. The sender of a message must always specify the receiver explicitly. The receiver however, has the possibility of mentioning the sender that it wants to communicate with, but it can also omit the indication of its communication partner, in which case it is willing to communicate with any sender that sends a message of the correct type. A message consists of a data value, which is transferred from the sender to the receiver. This data value can be a reference to an object or it can be an integer or boolean.

In order to describe by a program the unbounded number of objects in a system, we group them into *classes*. All objects in one class (the *instances* of that class) have the same structure of variables (each object has its own private variables, but the variables of all instances of a class have the same names and types) and they execute identical local processes. In this way a class can be considered as a blueprint for creating new instances.

Let us now give a formal definition of the language P. We assume as given a set C of *class names*, with typical element c . By this we mean that symbols like c , c' , c_1 , etc. will range over the set C of class names. (Appendix A.1 gives an overview of the sets used in this paper, together with their typical elements.) The set $C \cup \{\text{Int}, \text{Bool}\}$ of *data types*, with typical element d , we denote by C^+ . Here Int and Bool denote the types of the integers and booleans, respectively. For each $c \in C$ and $d \in C^+$ we assume $IVar_d^c$ to be the set of instance variables of type d in class c , with typical elements x_d^c and y_d^c . Such a variable x_d^c occurs in each object of class c and it can refer to objects of type d only. We assume that $IVar_d^c \cap IVar_{d'}^{c'} = \emptyset$ whenever $c \neq c'$ or $d \neq d'$. In cases where no confusion arises we omit the subscripts and superscripts.

Definition 2.1

We define the set Exp_d^c of expressions of type d in class c , with typical element e_d^c . Such an expression e_d^c can be evaluated by an object of class c and the object to which it refers will be of type d .

These expressions are defined as follows:

$$\begin{aligned}
 e_d^c ::= & \quad x_d^c \\
 & \quad | \quad \text{self} \quad \text{if } d = c \\
 & \quad | \quad \text{nil} \\
 & \quad | \quad \text{true} \mid \text{false} \quad \text{if } d = \text{Bool} \\
 & \quad | \quad n \quad \text{if } d = \text{Int} \\
 & \quad | \quad e_{1\text{Int}}^c + e_{2\text{Int}}^c \quad \text{if } d = \text{Int} \\
 & \quad | \quad \vdots \\
 & \quad | \quad e_{1d'}^c \doteq e_{2d'}^c \quad \text{if } d = \text{Bool}
 \end{aligned}$$

An expression e_d^c will be evaluated by a certain object α of class c . An expression of the form x denotes the value of the variable x that belongs to the object α . The expression self denotes the object α itself. The expression nil denotes no object at all. It can be used for every type, including Int and Bool . The symbols true and false stand for the corresponding values of type Bool . Every integer n can occur as an expression of type Int ; it simply denotes itself. We assume that the standard arithmetic and comparison operations on integers are available, but we list only the operator '+'. We assume that all these operations result in nil whenever an error occurs (e.g., division by zero or nil as an operand). Finally, for every type we have a test for equality. The expression $e_1 \doteq e_2$ evaluates to true whenever e_1 and e_2 denote the same object (or both denote no object, viz. nil). Note that in the programming language we put a dot over the equality sign (\doteq) to distinguish it from the equality sign we use in the metalanguage.

Definition 2.2

We next define the set Stat^c of statements in class c , with typical element S^c . These statements can be executed by an object of class c .

Statements can be of the following forms:

$$\begin{aligned}
 S^c ::= & \quad x_d^c := e_d^c \\
 & \quad | \quad x_d^c := \text{new}_d \quad \text{if } d \neq \text{Int}, \text{Bool} \\
 & \quad | \quad x_c^c ! e_d^c \\
 & \quad | \quad x_c^c ? y_d^c \\
 & \quad | \quad ? y_d^c \\
 & \quad | \quad S_1^c; S_2^c \\
 & \quad | \quad \text{if } e_{\text{Bool}}^c \text{ then } S_1^c \text{ else } S_2^c \text{ fi} \\
 & \quad | \quad \text{while } e_{\text{Bool}}^c \text{ do } S^c \text{ od}
 \end{aligned}$$

A statement S^c can be executed by an object of class c . The object executes the assignment statement $x := e$ by first evaluating the expression e at the right-hand side and then storing the result in its own variable x . The execution of the new-statement $x := \text{new}_d$ by the object α consists of creating a new object β of class d and making the variable x of the creator α refer to it. The instance variables of the new object β are initialized to nil and β will immediately start executing its local process. It is not possible to create new elements of the standard data types `Int` and `Bool`.

A statement $x_c^c!e_d^c$ is called an *output* statement and statements like $x_c^c?y_d^c$ and $?y_d^c$ are called *input* statements. Together they are called I/O statements. The execution of an output statement $x_{1c}^c!e_d^c$ by an object α is always synchronized with the execution of a corresponding input statement $x_{2c'}^c?y_d^c$ or $?y_d^c$ by another object β . Such a pair of input and output statements are said to *correspond* if all the following conditions are satisfied:

- The variable x_1 of the sending object α should refer to the receiving object β (therefore necessarily the type of the variable x_1 coincides with the class c' of β).
- If the input statement to be executed is of the form $x_{2c'}^c?y_d^c$, then the variable x_2 of the receiving object β should refer to the sending object α (again, this means that the type of the variable x_2 coincides with the class c of α).
- The type d of the expression e_d^c in the output statements should coincide with the type of the destination variable y_d^c in the input statement.

If an object tries to execute a I/O statement, but no other object is trying to execute a corresponding statement yet, it must wait until such a communication partner appears. If two objects are ready to execute corresponding I/O statements, the communication may take place. This means that the value of the expression e in the sending object α is assigned to the destination variable y in the receiving object β . When an object is ready to execute an input statement $?y$ there may be several objects ready to execute a corresponding output statement. One of them is chosen non-deterministically.

Statements are built up from these atomic statements by means of sequential composition, denoted by the semicolon ';', the conditional construct if-then-else-fi and the iterative construct while-do-od. The meaning of these constructs we shall assume to be known.

Definition 2.3

Finally we define the set *Prog* of *programs*, with typical element ρ , as follows:

$$\rho ::= \langle c_1 \leftarrow S_1^{c_1}, \dots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}} : S_n^{c_n} \rangle$$

Here we require that all the class names c_1, \dots, c_n are different. Furthermore we require for every variable x_d^c occurring in ρ that its type d is among $c_1, \dots, c_n, \text{Int}, \text{Bool}$ and that in every new-statement $x := \text{new}_d$ the type d of the newly created object is among c_1, \dots, c_{n-1} .

The first part of a program consists of a finite number of class definitions $c_i \leftarrow S_i$, which determine the local processes of the instances of the classes c_1, \dots, c_{n-1} . Whenever a new object of class c_i is created, it will begin to execute the corresponding statement S_i . The second part specifies the local process S_n of the *root class* c_n . The execution of a program starts with the creation of a single instance of this root class, the *root object*, which begins executing the statement S_n . This root object can create other objects in order to establish parallelism. Due to the above restriction on the types of new-statements, the root object will always be the only instance of its class.

An example program

We illustrate the programming language by giving a program that generates the prime numbers up to a certain constant n . The program uses the sieve method of Eratosthenes. It consists of two classes. The class G (for 'generator') describes the behaviour of the root object, which consists of generating the natural numbers from 2 to n and sending them to an object of the other class P. The objects of the class P (for 'prime sieve') essentially form a chain of filters. Each of these objects remembers the first number it is sent; this will always be a prime. From the numbers it receives subsequently, it will simply discard the ones that are divisible by its local prime number, and it will send the others to the next P object in the chain.

The class G makes use of two instance variables: f (for 'first') of type P and c (for 'count') of type Int (note that n is not an instance variable but an integer constant). The class P has three instance variables: m ('my prime') and b ('buffer') of type Int and l ('link') of type P. Here is the complete program:

```

⟨P ← ?m;
  if m ≠ nil
  then l := new;
    ?b;
    while b ≠ nil
    do if m ≠ b then l ! b; fi;
      ?b
    od;
    l ! b
  fi
: f := new; c := 2;
  while c ≤ n do f ! c; c := c + 1 od;
  f ! nil⟩

```

3 The assertion language

In this section we define two different assertion languages. An *assertion* describes the state of (a part of) the system at one specific point during its execution. The first assertion language describes the *internal state* of a single object. This is called the *local* assertion language. It will be used in the local proof system. The other one, the *global* assertion language, describes a whole system of objects. It will be used in the intermediate and global proof systems.

3.1 The local assertion language

First we introduce a new kind of variables: For $d = \text{Int}, \text{Bool}$, let LogVar_d be an infinite set of *logical variables* of type d , with typical element z_d . We assume that these sets are disjoint from the other sets of syntactic entities. Logical variables do not occur in a program, but only in assertions.

Definition 3.1

The set LExp_d^c of *local expressions* of type d in class c , with typical element l_d^c , is

defined as follows:

$$\begin{aligned}
 l_d^c &::= z_d \\
 &| x_d^c \\
 &| \text{self} \quad \text{if } d = c \\
 &| \text{nil} \\
 &| n \quad \text{if } d = \text{Int} \\
 &| \text{true} \mid \text{false} \quad \text{if } d = \text{Bool} \\
 &| l_{1\text{Int}}^c + l_{2\text{Int}}^c \quad \text{if } d = \text{Int} \\
 &| \vdots \\
 &| l_{1d'}^c \doteq l_{2d'}^c \quad \text{if } d = \text{Bool}
 \end{aligned}$$

Definition 3.2

The set $LAss^c$ of *local assertions* in class c , with typical element p^c , is defined as follows:

$$\begin{aligned}
 p^c &::= l_{\text{Bool}}^c \\
 &| \neg p^c \\
 &| p_1^c \wedge p_2^c \\
 &| \exists z_d p^c \quad (d = \text{Bool}, \text{Int})
 \end{aligned}$$

We shall regard other logical connectives ($\vee, \rightarrow, \forall$) as abbreviations for combinations of the above ones.

An example of a local assertion is

$$\neg(b \doteq \text{nil}) \rightarrow \forall i(2 \leq i \wedge i \leq m \rightarrow \neg(i \mid b)),$$

which we might abbreviate to $b \neq \text{nil} \rightarrow \forall i(2 \leq i \leq m \rightarrow i \nmid b)$. Here i is a logical variable of type Int . (Most of our examples will be in the context of the program at the end of section 2.)

Local expressions l_d^c and local assertions p^c are evaluated with respect to the local state of an object of class c , determining the values of its instance variables, plus a logical environment, which assigns values to the logical variables. Therefore they talk about this single object in isolation. It is important to note that we allow only logical variables ranging over integers and booleans to occur in local expressions, so that only quantification over integers and booleans is possible. (By the way, the value nil is *not* included in the range of quantifications.) As we shall see below, quantification over other types would require knowledge of the set of existing objects, which is not available locally.

3.2 The global assertion language

Next we define the global assertion language. Since in the global assertion language we also want to quantify over objects of any class $c \in C^+$, we now need for every $c \in C$ a new set $LogVar_c$ of logical variables of type c , with typical element z_c . To be able to describe interesting properties of pointer structures we also introduce logical variables ranging over *finite sequences* of objects. To do so we first introduce for every $d \in C^+$ the type d^* of finite sequences of objects of type d . We define $C^* = \{d^* : d \in C^+\}$ and take $C^\dagger = C^+ \cup C^*$, with typical element a . Now in addition we assume for every $d \in C^+$ the set $LogVar_{d^*}$ of logical variables of type d^* , which range over finite sequences of elements of type d . Therefore we now have a set $LogVar_a$ of logical variables of type a for every $a \in C^\dagger$.

Definition 3.3

The set $GExp_a$ of *global expressions* of type a , with typical element g_a , is defined as follows:

$g_a ::=$	z_a	
	nil	
	n	if $a = \text{Int}$
	true false	if $a = \text{Bool}$
	$g_c.x_d^c$	if $a = d$
	$ g_{d^*} $	if $a = \text{Int}$
	$g_{d^*} : g_{\text{Int}}$	if $a = d$
	$g_{1_{\text{Int}}} + g_{2_{\text{Int}}}$	if $a = \text{Int}$
	\vdots	
	if g_{Bool} then g_{1_a} else g_{2_a} fi	
	$g_{1_d} \doteq g_{2_d}$	if $a = \text{Bool}$

A global expression is evaluated with respect to a complete system of objects plus a logical environment. A complete system of objects consists of a set of existing objects together with their local states. For sequence types the expression nil denotes the empty sequence. The expression $g.x$ denotes the value of the variable x of the object denoted by g . Note that in this global assertion language we must explicitly specify the object of which we want to access the internal data. $|g|$ denotes the length of the sequence denoted by g . The expression $g_1 : g_2$ denotes the n th element of the sequence denoted by g_1 , where n is the value of g_2 (if g_2 is less than 1 or greater than $|g_1|$, the result is nil.) The conditional expression if g_0 then g_1 else g_2 fi is introduced to facilitate the handling of aliasing (see definition 4.3). If the condition is nil, then the result of the conditional expression is nil, too.

Definition 3.4

The set $GAss$ of *global assertions*, with typical element P , is defined as follows:

$$\begin{aligned}
 P &::= g_{Bool} \\
 &| \neg P \\
 &| P_1 \wedge P_2 \\
 &| \exists z_a P
 \end{aligned}$$

Again, other logical connectives are regarded as abbreviations.

Quantification over (sequences of) integers and booleans is interpreted as usual. However, quantification over (sequences of) objects of some class c is interpreted as ranging only over the *existing* objects of that class, i.e., the objects that have been created up to the current point in the execution of the program. For example, the assertion $\exists z_c \text{true}$ is false in some state iff there are no objects of class c in this state. More interestingly, the assertion

$$\forall p \exists s. (s : 1 \doteq g.f \wedge s : |s| \doteq p \wedge \forall i (1 \leq i < |s| \rightarrow (s : i).l \doteq s : (i+1)))$$

expresses that every object of class P is a member of the l -linked chain that starts with $g.f$ (where g is the generator object).

Next we define a transformation of a local expression or assertion to a global one. This transformation will be used to verify the assumptions made in the local proof system about the I/O and new-statements. These assumptions are formulated in the local language. As the reasoning in the cooperation test uses the global assertion language we have to transform these assumptions from the local language to the global one.

Definition 3.5

Given a local expression l_d^c and a global expression g_c we define a global expression $l_d^c \downarrow g_c$. This expression denotes the result of evaluating the local expression l in the object denoted by the global expression g . The definition proceeds by induction on the complexity of the local expression l :

$$\begin{aligned}
 l \downarrow g &= l && \text{if } l = z, \text{nil}, n, \text{true}, \text{false} \\
 x \downarrow g &= g.x \\
 \text{self} \downarrow g &= g \\
 (l_1 + l_2) \downarrow g &= (l_1 \downarrow g) + (l_2 \downarrow g) \\
 (l_1 \doteq l_2) \downarrow g &= (l_1 \downarrow g) \doteq (l_2 \downarrow g)
 \end{aligned}$$

For a local assertion p^c we define the global assertion $p^c \downarrow g_c$ as follows:

$$\begin{aligned}
 (\neg p) \downarrow g &= (\neg p \downarrow g) \\
 (p_1 \wedge p_2) \downarrow g &= (p_1 \downarrow g) \wedge (p_2 \downarrow g) \\
 (\exists z p) \downarrow g &= \exists z (p \downarrow g)
 \end{aligned}$$

As an example, note that $(b \neq \text{nil} \rightarrow \forall i(2 \leq i \leq m \rightarrow i \wedge b)) \downarrow p$ is equal to $p.b \neq \text{nil} \rightarrow \forall i(2 \leq i \leq p.m \rightarrow i \wedge p.b)$.

3.3 Correctness formulas

In this section we define how we specify an object and a complete system of objects, using the formalism of Hoare triples. We start with the specification of an object.

Definition 3.6

We define a *local correctness formula* to be of the following form:

$$\{p^c\}S^c\{q^c\}.$$

Here the assertion p is called the *precondition* and the assertion q is called the *postcondition*. The meaning of such a correctness formula is described informally as follows:

Every terminating execution of S by an object of class c starting from a state satisfying p will end in a state satisfying q .

As said before, reasoning about the local correctness of an object will be done relative to assumptions concerning those parts of its local process that depend on the environment. These parts are called *bracketed sections*:

Definition 3.7

A bracketed section is a construct of the form $\langle S_1^c; S^c; S_2^c \rangle$, where S^c denotes an I/O statement or a new-statement, and in S_1 and S_2 neither I/O statements nor new-statements occur (note that S_1 and S_2 can each be composed of several statements by means of sequential compositions, conditionals, or loops). Furthermore, if $S = x := \text{new}$ then the variable x must not occur at the left-hand side of an assignment occurring in S_2 . The additional condition on the variable x , which is used to store the identity of the new object, is to ensure that after the execution of the corresponding bracketed section this new object is still referred to by the variable x .

Next we define intermediate correctness formulas, which describe the behaviour of an object executing a bracketed section containing a new-statement or a communication between two objects.

Definition 3.8

An *intermediate correctness formula* can have one of the following two forms:

- $\{P\}(z_c, S^c)\{Q\}$, where S is a local statement or a bracketed section containing a new-statement.
- $\{P\}(z_{c_1}, S_1^{c_1}) \parallel (z'_{c_2}, S_2^{c_2})\{Q\}$, where z and z' are distinct logical variables and $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are bracketed sections that contain I/O statements.

The logical variables z_c , z_{c_1} , and z'_{c_2} in the above constructs denote the objects that are considered to be executing the corresponding statements. More precisely, the meaning of the intermediate correctness formula $\{P\}(z, S)\{Q\}$ is as follows:

Every terminating execution of the bracketed section S by the object denoted by the logical variable z starting in a (global) state satisfying P ends in a (global) state satisfying Q .

The meaning of the second form of intermediate correctness formula, $\{P\}(z, S_1) \parallel (z', S_2)\{Q\}$, can be described as follows:

Every terminating parallel execution of the bracketed section S_1 by the object denoted by the logical variable z and of S_2 by the object denoted by z' starting in a (global) state satisfying P will end in a (global) state satisfying Q .

Finally, we have global correctness formulas, which describe a complete system:

Definition 3.9

A global correctness formula is of the form

$$\{p^c \downarrow z_c\} \rho^c \{Q\}$$

The variable z in such a global correctness formula denotes the root object. Initially this root object is the only existing object, so it is sufficient for the precondition of a complete system to describe only its local state. We obtain such a precondition by transforming some local assertion p to a global one. On the other hand, the final state of an execution of a complete system is described by an arbitrary global assertion. The meaning of the global correctness formula $\{p \downarrow z\} \rho \{Q\}$ can be rendered as follows:

If the execution of the program ρ starts with a root object denoted by z that satisfies the local assertion p and no other objects, and if moreover this execution terminates, then the final state will satisfy the global assertion Q .

4 The proof system

The proof system we present consists of three levels. The first level, called the *local* proof system, allows us to reason about the correctness of a single object. Testing the assumptions that are introduced at this first level to deal with I/O statements and new-statements, is done at the second level, which is called the *intermediate* proof system. The third level, the *global* proof system, formalizes the reasoning about a complete system.

4.1 The local proof system

The proof system for local correctness formulas is similar to the usual system for sequential programs.

Definition 4.1

The local proof system consists of the following axiom and rules:

Assignment:

$$\{p[e/x]\}x := e\{p\} \quad (\text{LASS})$$

Sequential composition:

$$\frac{\{p\}S_1\{r\}, \quad \{r\}S_2\{q\}}{\{p\}S_1; S_2\{q\}} \quad (\text{LSC})$$

Conditional:

$$\frac{\{p \wedge e\}S_1\{q\}, \quad \{p \wedge \neg e\}S_2\{q\}}{\{p\}\text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}\{q\}} \quad (\text{LCOND})$$

Iteration:

$$\frac{\{p \wedge e\}S\{p\}}{\{p\}\text{while } e \text{ do } S \text{ od}\{p \wedge \neg e\}} \quad (\text{LIT})$$

Consequence:

$$\frac{p \rightarrow p_1, \quad \{p_1\}S\{q_1\}, \quad q_1 \rightarrow q}{\{p\}S\{q\}} \quad (\text{LCR})$$

Reasoning about new-statements and I/O statements is done by introducing assumptions about them, where assumptions have the form of local correctness formulas.

The substitution operation $[e/x]$ occurring in the above assignment axiom is the ordinary substitution, i.e., literal replacement of every occurrence of the variable x by the expression e . This works because at this level we have no aliasing, i.e., it is not possible that different local expressions denote the same variable. In the intermediate and global proof systems we shall have to take special measures to deal with aliasing.

4.2 The intermediate proof system

In this section we present the proof system for intermediate correctness formulas.

4.2.1 The assignment axiom

Definition 4.2

The assignment axiom in the intermediate proof system has the form

$$\{P[e \downarrow z/z.x]\}(z, x := e)\{P\} \quad (\text{IASS})$$

where $x \in IVar_d^c$, $e \in Exp_d^c$, $z \in LogVar_c$, and $P \in GAss$.

First note that we have to transform the expression e to the global expression $e \downarrow z$ and substitute this latter expression for $z.x$ because we consider the execution of the assignment $x := e$ by the object denoted by z . Furthermore we have to pay special attention to the substitution $[e \downarrow z/z.x]$ because the usual substitution does not take into account that there are many different global expressions that may denote the same variable $z.x$. This problem is solved by the following definition.

Definition 4.3

Given a global expression g_d , a logical variable z_c and an instance variable x_d^c , we define for any global expression g' the substitution $g'[g/z.x]$ by induction on the complexity of g' as follows.

$$\begin{aligned} g'[g/z.x] &= g' && \text{if } g' = z', n, \text{nil}, \text{self}, \text{true}, \text{false} \\ (g'.y)[g/z.x] &= g'[g/z.x].y && \text{if } y \neq x \\ (g'.x)[g/z.x] &= \text{if } g'[g/z.x] = z \text{ then } g \text{ else } g'[g/z.x].x \text{ fi} \\ &\vdots \\ (g_1 \doteq g_2)[g/z.x] &= (g_1[g/z.x]) \doteq (g_2[g/z.x]) \end{aligned}$$

The omitted cases are defined directly from the application of the substitution to the subexpressions, like the last one. This substitution operation is generalized to global assertions in a straightforward manner, with the notation $P[g/z.x]$.

As an example, consider the postcondition $\forall i(1 \leq i \leq |s| \rightarrow (s : i).x \doteq i)$ for the statement $x := y + 1$, executed by the object denoted by z . The precondition given by the axiom (IASS) is

$$\forall i(1 \leq i \leq |s| \rightarrow \text{if } s : i \doteq z \text{ then } z.y + 1 \text{ else } (s : i).x \doteq i).$$

The best way to justify definition 4.3 is by comparing it to the ordinary substitution $[e/x]$, which is used in the local assignment axiom (LASS). The essential property of this substitution is that the substituted expression, evaluated in the state before the assignment, has the same value as the original expression in the state after the assignment. Quasi-formally, we could write this as

$$\llbracket [e/x] \rrbracket(\sigma) = \llbracket e \rrbracket(\sigma')$$

where σ' is the state that results from executing the assignment $x := e$ in the state σ . We could say that the substitution is a way of predicting the value that an expression or assertion will have after performing an assignment.

It is easy to prove that the substitution operation defined in definition 4.3 has exactly the same property:

$$\llbracket [g/z.x] \rrbracket(\sigma) = \llbracket g \rrbracket(\sigma')$$

and

$$\sigma \models P[g/z.x] \iff \sigma' \models P$$

where σ' is the state that results from σ by changing the value of the variable x in the object denoted by z to the value $\llbracket g \rrbracket(\sigma)$ that results from evaluating the expression g in the state σ .

The most important aspect of this substitution is certainly the conditional expression that turns up when we are dealing with an expression of the form $g'.x$. This is necessary because a certain form of aliasing is possible: After the assignment it may be the case that g' refers to the same object as the logical variable z , so that $g'.x$ is the same variable as $z.x$, which has the value $\llbracket g \rrbracket(\sigma)$. It is also possible that, after the assignment, g' does not refer to the object denoted by z , so that the value of $g'.x$ does not change. Since we can not decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides dynamically.

The notation $[./.]$ may seem overloaded now, but it is always possible to determine the required operation from the form of the arguments (see appendix A.2).

4.2.2 The creation of new objects

Definition 4.4

We describe the new-statement by the following axiom of the intermediate proof system:

$$\{P[z'/z.x][\text{new}/z']\}(z, x := \text{new})\{P\} \quad (\text{NEW})$$

where $x \in IVar_d^c$, $d \in C$, $z \in LogVar_c$, $z' \in LogVar_d$, $z \neq z'$, $P \in GAss$, and z' does not occur in P .

This axiom reflects the fact that we can view the execution of the statement $x := \text{new}$ as consisting of two steps: first the new object is created and temporarily stored in the logical variable z' , and then the value of this logical variable z' is assigned to the instance variable x of the object denoted by z . The second step is dealt with by the substitution $[z'/z.x]$ of definition 4.3, which takes care of all possible complications of aliasing.

For the creation of a new object we have to define another substitution operation $[\text{new}/z]$. This is complicated by the fact that the newly created object does not exist in the state just before its creation, so that in this state we can not refer to it. Fortunately, we do need this substitution for all possible global expressions, but primarily for assertions, and in an assertion the logical variable z can essentially occur in only two contexts: either one of its instance variables is referenced, or it is compared for equality with another expression. In both cases we can predict the outcome without having to refer to the new object.

Definition 4.5

Let $d \in C$ and $z \in \text{LogVar}_d$. For certain global expressions g we define $g[\text{new}/z]$ by induction on the complexity of g . We only list the interesting cases.

$z'[\text{new}/z]$	$= z'$	if $z' \neq z$
$z[\text{new}/z]$	is undefined	
$g[\text{new}/z]$	$= g$	if $g = n, \text{nil}, \text{self}, \text{true}, \text{false}$
$(z'.x)[\text{new}/z]$	$= z'.x$	if $z' \neq z$
$(z.x)[\text{new}/z]$	$= \text{nil}$	
$(g.y.x)[\text{new}/z]$	$= (g.y)[\text{new}/z].x$	
$(g_1 \doteq g_2)[\text{new}/z]$	$= g_1[\text{new}/z] \doteq g_2[\text{new}/z]$	if $g_1, g_2 \neq z$, if ... fi
$(g_1 \doteq z)[\text{new}/z]$	$= \text{false}$	if $g_1 \neq z$, if ... fi
$(z \doteq g_2)[\text{new}/z]$	$= \text{false}$	if $g_2 \neq z$, if ... fi
$(z \doteq z)[\text{new}/z]$	$= \text{true}$	

Here we have ignored the conditional expressions (these can be removed before substituting). In all the other cases not listed above, the substitution $[\text{new}/z]$ can be applied to an expression by applying it to the constituent expressions. In this way $g[\text{new}/z]$ is defined for all global expressions g except for z itself (and for certain conditional expressions). Again it is rather easy to prove that this substitution has the desired property:

$$\llbracket g[\text{new}/z] \rrbracket(\sigma) = \llbracket g \rrbracket(\sigma')$$

where σ' can be obtained from σ by creating a new object (with all its variables initialized to nil) and storing it in the variable z .

Definition 4.6

We define $P[\text{new}/z_c]$ by induction on the complexity of the global assertion P .

$$\begin{aligned}
g_{\text{Bool}}[\text{new}/z_c] & \quad \text{as in definition 4.5} \\
(\neg P)[\text{new}/z_c] & = \neg(P[\text{new}/z_c]) \\
(P_1 \wedge P_2)[\text{new}/z_c] & = (P_1[\text{new}/z_c] \wedge P_2[\text{new}/z_c]) \\
(\exists z_a P)[\text{new}/z_c] & = \exists z_a (P[\text{new}/z_c]) & \text{if } a \neq c, c^* \\
(\exists z'_c P)[\text{new}/z_c] & = \exists z'_c (P[\text{new}/z_c]) \vee (P[z_c/z'_c][\text{new}/z_c]) & \text{if } z'_c \neq z_c \\
(\exists z_{c^*} P)[\text{new}/z_c] & = \exists z_{c^*} \exists z_{\text{Bool}^*} (|z_{c^*}| \doteq |z_{\text{Bool}^*}| \wedge P[z_{\text{Bool}^*}, z_c/z_{c^*}][\text{new}/z_c])
\end{aligned}$$

Here we assume that z_{Bool^*} does not occur in P . The substitution $[z_{\text{Bool}^*}, z_c/z_{c^*}]$ will be defined in definition 4.7.

The case of quantification over the type c of the newly created object can be explained as follows: Remember that we interpret the result of the substitution in a state in which the object denoted by z does not yet exist. In the first part $\exists z'(P[\text{new}/z])$ of the substituted formula the bound variable z' thus ranges over all the old objects. In the second part the object to be created is dealt with separately. This is done by first substituting z for z' (expressing that the quantified variable z' takes the same value as the variable z) and then applying the substitution $[\text{new}/z]$ (note that simply applying $[\text{new}/z']$ does not give the right result in the case that z occurs in P). Together the two parts of the substituted formula express quantification over the whole range of existing objects in the new state.

Let us consider, for example, the statement $x := \text{new}$, to be executed by the object indicated by the logical variable z , and the given postcondition

$$\forall v (v.x \neq \text{nil} \rightarrow v.y \doteq 1).$$

In order to determine the corresponding precondition given by the axiom (NEW), we first apply the substitution $[z'/z.x]$, which leads to

$$\forall v (\text{if } v \doteq z \text{ then } z' \text{ else } v.x \neq \text{nil} \rightarrow v.y \doteq 1).$$

We can remove the conditional expression by taking the equivalent assertion

$$\forall v ((v \doteq z \wedge z' \neq \text{nil}) \vee (v \neq z \wedge v.x \neq \text{nil}) \rightarrow v.y \doteq 1).$$

Now to this we apply the substitution $[\text{new}/z']$, resulting in

$$\begin{aligned}
& \forall v ((v \doteq z \wedge \text{true}) \vee (v \neq z \wedge v.x \neq \text{nil}) \rightarrow v.y \doteq 1) \\
& \wedge ((\text{false} \wedge \text{true}) \vee (\text{true} \wedge \text{nil} \neq \text{nil}) \rightarrow \text{nil} \doteq 1),
\end{aligned}$$

which can be simplified to

$$\forall v (v \doteq z \vee v.x \neq \text{nil} \rightarrow v.y \doteq 1).$$

For quantification about sequences of objects of class c , we need a slightly more elaborate mechanism. The sequences over which we quantify in the old state cannot contain the new object as an element. Therefore we use two sequence variables z_c^\bullet and z_{Bool}^\bullet to code one sequence of objects in the new state. The idea of the substitution operation $[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet]$ is that at the places where z_{Bool}^\bullet yields true, the value of the coded sequence is the newly created object, here denoted by the variable z_c . Where z_{Bool}^\bullet yields false, the value of the coded sequence is the same as the value of z_c^\bullet and where z_{Bool}^\bullet delivers nil the sequence also yields nil. This is formalized in the following definition.

Definition 4.7

For certain global expressions g we define $g[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet]$ as follows (again we list only the interesting cases):

$$\begin{aligned}
 z_c^\bullet[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & \quad \text{is undefined} \\
 z[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & = z \quad \text{if } z \neq z_c^\bullet \\
 g[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & = g \quad \text{if } g = n, \text{nil}, \text{self}, \text{true}, \text{false} \\
 (g.x)[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & = g[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet].x \\
 (z_c^\bullet : g)[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & = \text{if } z_{\text{Bool}}^\bullet : (g[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet]) \\
 & \quad \text{then } z_c \\
 & \quad \text{else } z_c^\bullet : (g[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet]) \\
 & \quad \text{fi} \\
 (g_1 : g_2)[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & = g_1[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] : g_2[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] \quad \text{if } g_1 \neq z_c^\bullet \\
 (|z_c^\bullet|)[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & = |z_c^\bullet| \\
 (|g|)[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet] & = |g[z_{\text{Bool}}^\bullet, z_c/z_c^\bullet]| \quad \text{if } g \neq z_c^\bullet
 \end{aligned}$$

The generalization of the above to other global expressions and to global assertions is straightforward.

Again we have the desired property:

$$\sigma \models P[\text{new}/z] \iff \sigma' \models P$$

where σ' can be obtained from σ by creating a new object (with all its variables initialized to nil) and storing it in the variable z .

We illustrate the last definition by another example of the use of axiom (NEW). Again we take the statement $x := \text{new}$ executed by the object in z , but this time the postcondition is $\exists s \forall p \exists i (s : i \doteq p)$. Applying the substitution $[z'/z.x]$ leaves this

assertion unchanged, so we can directly apply the substitution $[new/z']$. We calculate in the following few steps:

$$\begin{aligned}
& (\exists s \forall p \exists i (s : i \doteq p)) [new/z'] \\
&= \exists s \exists b_{\text{Bool}} \bullet |s| \doteq |b| \wedge (\forall p \exists i (s : i \doteq p)) [b, z'/s] [new/z'] \\
&= \exists s \exists b |s| \doteq |b| \wedge (\forall p \exists i (\text{if } b : i \text{ then } z' \text{ else } s : i \text{ fi } \doteq p)) [new/z'] \\
&\equiv \exists s \exists b |s| \doteq |b| \wedge (\forall p \exists i (b : i \wedge z' \doteq p) \vee (\neg b : i \wedge s : i \doteq p)) [new/z'] \\
&= \exists s \exists b |s| \doteq |b| \wedge \forall p \exists i ((b : i \wedge \text{false}) \vee (\neg b : i \wedge s : i \doteq p)) \\
&\quad \wedge \exists i (b : i \wedge \text{true}) \vee (\neg b : i \wedge \text{false}) \\
&\equiv \exists s \exists b (|s| \doteq |b| \wedge \forall p \exists i (\neg b : i \wedge s : i \doteq p) \wedge \exists i (b : i))
\end{aligned}$$

Here \equiv denotes semantic equivalence, which means that the assertions on both sides will have the same truth value in every environment. (By the way, in any state that can occur in the execution of a P program, the above assertions will be true, since there is only a finite number of objects of any class.)

4.2.3 Communication

Now we define an axiom and some rules which together describe the communication between objects.

Definition 4.8

Let $S_1 \in \text{Stat}^c$ be of the form $x?y$ or $?y$ and $S_2 = x'!e \in \text{Stat}^{c'}$ such that $x \in \text{IVar}_c^c$, $x' \in \text{IVar}_c^{c'}$, and the variable y is of the same type as the expression e . Such a pair of I/O statements are said to *match*. Furthermore let $z \in \text{LogVar}_c$ and $z' \in \text{LogVar}_{c'}$ be two distinct variables and let P be a global assertion. Then the following is an instance of the communication axiom:

$$\{P[e \downarrow z'/z.y]\}(z, S_1) \parallel (z', S_2)\{P\} \quad (\text{COMM})$$

Note that the communication is described by a substitution that expresses the assignment of $e \downarrow z'$ to $z.y$ (definition 4.3).

The following two rules show how one can use the information about the relationship between the receiver and the sender which must hold for the communication to take place, that is, the sender must refer to the receiver, and if the receiver executes an input statement of the form $x?y$, it must refer to the sender.

Definition 4.9

Let $x?y, ?y \in \text{Stat}^c$ and $x!e \in \text{Stat}^{c'}$ be such that both input statements match with the output statement. Furthermore let $z \in \text{LogVar}_c$ and $z' \in \text{LogVar}_{c'}$ be two distinct variables. Then we have the following two rules:

$$\frac{\{P \wedge z.x \doteq z' \wedge z' \neq \text{nil} \wedge z'.x' \doteq z \wedge z \neq \text{nil} \wedge R\}(z, x?y) \parallel (z', x!e)\{Q\}}{\{P\}(z, x?y) \parallel (z', x!e)\{Q\}} \quad (\text{SR1})$$

and

$$\frac{\{P \wedge z'.x' \doteq z \wedge z \neq \text{nil} \wedge R\}(z, ?y) \parallel (z', x!e)\{Q\}}{\{P\}(z, ?y) \parallel (z', x!e)\{Q\}} \quad (\text{SR2})$$

where $R = (z \neq z')$ if $c = c'$ and $R = \text{true}$ otherwise.

The following rule describes the independent parallel execution of two objects.

Definition 4.10

Suppose that $S_1 \in \text{Stat}^c$ and $S_2 \in \text{Stat}^{c'}$ do not contain any I/O or new-statements. Furthermore let $z \in \text{LogVar}_c$ and $z' \in \text{LogVar}_{c'}$ be two distinct variables. Then we have the following rule:

$$\frac{\{P\}(z, S_1)\{R\}, \quad \{R\}(z', S_2)\{Q\}}{\{P\}(z, S_1) \parallel (z', S_2)\{Q\}} \quad (\text{PAR1})$$

Note that this rule models the fact that the parallel execution of two local computations can be sequentialized. The next rule takes care of the case where the two bracketed sections do contain I/O statements.

Definition 4.11

Let $\langle S_1; S; S_2 \rangle \in \text{Stat}^c$ and $\langle S'_1; S'; S'_2 \rangle \in \text{Stat}^{c'}$ be two bracketed sections containing I/O statements. Furthermore let $z \in \text{LogVar}_c$ and $z' \in \text{LogVar}_{c'}$ be two distinct variables. Then we have the rule

$$\frac{\{P\}(z, S_1) \parallel (z', S'_1)\{P_1\}, \quad \{P_1\}(z, S) \parallel (z', S')\{Q_1\}, \quad \{Q_1\}(z, S_2) \parallel (z', S'_2)\{Q\}}{\{P\}(z, S_1; S; S_2) \parallel (z', S'_1; S'; S'_2)\{Q\}} \quad (\text{PAR2})$$

Finally, the intermediate proof system contains rules for sequential composition (ISC), the conditional statement (ICOND), the iterative construct (IIT), and a consequence rule (ICR). These are straightforward modifications of the corresponding rules of the local proof system, so we only give the iteration rule as an example.

Definition 4.12

Let $\text{while } t \text{ do } S \text{ od} \in \text{Stat}^c$ and $z \in \text{LogVar}_c$. Then we have the following rule:

$$\frac{\{P \wedge e \downarrow z\}(z, S)\{Q\}}{\{P\}(z, \text{while } e \text{ do } S \text{ od})\{P \wedge \neg e \downarrow z\}} \quad (\text{IIT})$$

4.3 The global proof system

In this section we describe the global proof system. Two bracketed sections containing I/O statements are said to *match* if the corresponding I/O statements match. A program ρ is said to be *bracketed* if every I/O statement and new-statement occurs in a bracketed section. An *assumption* is defined to be a local correctness assertion about a bracketed section. Given a set A of assumptions, the construct $A \vdash \{p^c\}S^c\{q^c\}$ denotes the derivability of the local correctness formula $\{p^c\}S^c\{q^c\}$ from the local proof system using the assumptions of A as additional axioms. Now we are ready to define the notion of the cooperation test:

Definition 4.13

Let $\rho = \langle c_1 \leftarrow S^{c_1}, \dots, c_{n-1} \leftarrow S^{c_{n-1}} : S^{c_n} \rangle$ be bracketed. Furthermore, for each k , $1 \leq k \leq n$, let A^{c_k} denote a set of local correctness formulas about the bracketed sections occurring in S^{c_k} such that $A^{c_k} \vdash \{p^{c_k}\}S^{c_k}\{q^{c_k}\}$. Finally, let I be some global assertion, which we shall call the *global invariant*. Now we say that the *cooperation test* holds, notation $\text{Coop}(A^{c_1}, \dots, A^{c_n}, I, \rho)$, if the following conditions are fulfilled:

1. The invariant I does not contain any instance variable that occurs at the left-hand side of an assignment outside a bracketed section.
2. Let $\langle S_1 \rangle$ and $\langle S_2 \rangle$ be two matching bracketed sections such that $\{p_1\}S_1\{q_1\} \in A^{c_i}$ and $\{p_2\}S_2\{q_2\} \in A^{c_j}$, where $1 \leq i \leq n-1$, $1 \leq j \leq n$. Furthermore let $z \in \text{LogVar}_{c_i}$ and $z' \in \text{LogVar}_{c_j}$ be two new distinct variables. Then we require

$$\vdash \{I \wedge p_1 \downarrow z \wedge p_2 \downarrow z'\}(z, S_1) \parallel (z', S_2)\{I \wedge q_1 \downarrow z \wedge q_2 \downarrow z'\}.$$

3. Let $\langle S \rangle$ be a bracketed section containing the new-statement $x := \text{new}_{c_j}$ such that $\{p_1\}S\{q_1\} \in A^{c_i}$. Furthermore let $z \in \text{LogVar}_{c_i}$ be a new variable. Then we require that

$$\vdash \{I \wedge p_1 \downarrow z\}(z, S)\{I \wedge q_1 \downarrow z \wedge p^{c_j} \downarrow z.x\}.$$

4. Let z_{c_n} be a new variable. Then the following assertion should hold:

$$p^{c_n} \downarrow z_{c_n} \wedge \forall z'_{c_n} (z'_{c_n} \doteq z_{c_n}) \wedge \bigwedge_{1 \leq i < n} (\forall z_{c_i} \text{ false}) \rightarrow I.$$

The syntactic restriction in clause 1 on occurrences of variables in the global invariant I implies the invariance of this assertion over those parts of the program that are not contained in a bracketed section. The clauses 2 and 3 imply, among others, the invariance of the global invariant over the bracketed sections.

This global invariant expresses some invariant properties of the global states arising during a computation of ρ . These properties are invariant in the sense that they hold whenever the program counter of every existing object is at a location outside a bracketed section. The above method to prove the invariance of the global invariant is based on the following semantical property of bracketed sections: Every computation of ρ can be rearranged such that at every time there is at most one object executing a bracketed section containing a new-statement, and an object is allowed to enter a bracketed section containing a I/O statement only if there is at most one other object executing a bracketed section.

Clause 2 establishes the cooperation between two arbitrary matching assumptions, where two assumptions are said to match if their corresponding bracketed sections contain matching I/O statements. Note that since there exists only one object of class c_n , the root object, we do not have to apply the cooperation test between two matching assumptions of the set A^{c_n} .

Clause 3 discharges assumptions about bracketed sections containing new-statements. Additionally the truth of the precondition of the local process of the new object is established. Note that by definition of a bracketed section we know that the variable x of the new-statement cannot occur as the left-hand side of an assignment after the new-statement. Therefore it refers to the newly created object immediately after the execution of such a bracketed section.

Clause 4 establishes the truth of the global invariant in the initial state. Note that the assertion $\forall z_c \text{ false}$ expresses that there exist no objects of class c . The assertion $\forall z'_{c_n} (z'_{c_n} \doteq z_{c_n})$ expresses that there exists precisely one object of class c_n .

In the following definitions, let $\rho = \langle c_1 \leftarrow S^{c_1}, \dots, c_{n-1} \leftarrow S^{c_{n-1}} : S^{c_n} \rangle$.

Definition 4.14

The program rule of the global proof system has the following form:

$$\frac{Coop(A^{c_1}, \dots, A^{c_n}, I, \rho) \quad A^{c_k} \vdash \{p^{c_k}\} S^{c_k} \{q^{c_k}\}, 1 \leq k \leq n}{\{p^{c_n} \downarrow z_{c_n}\} \rho \{I \wedge q^{c_n} \downarrow z_{c_n} \wedge \bigwedge_{1 \leq i < n} \forall z_{c_i} q^{c_i} \downarrow z_{c_i}\}} \quad (\text{PR})$$

Note that in the conclusion of the program rule (PR) we take as precondition the precondition of the local process of the root object because initially only this object exists. The postcondition consists of a conjunction of the global invariant, the assertion $q^{c_n} \downarrow z_{c_n}$ expressing that the final local state of the root object is characterized by the local assertion q^{c_n} , and the assertions $\forall z_{c_i} q^{c_i} \downarrow z_{c_i}$, which express that the final local state of every object of class c_i is characterized by the local assertion q^{c_i} .

Definition 4.15

We have the following consequence rule for programs:

$$\frac{p^{c_n} \rightarrow p_1^{c_n}, \quad \{p_1^{c_n} \downarrow z_{c_n}\} \rho \{Q_1\}, \quad Q_1 \rightarrow Q}{\{p^{c_n} \downarrow z_{c_n}\} \rho \{Q\}} \quad (\text{PCR})$$

Definition 4.16

Furthermore, we have a rule for *auxiliary variables*: For a given program ρ and a (global) postcondition Q , let Aux be a set of instance variables such that

- for any assignment $x := e$ occurring in ρ we have that $IVar(e) \cap Aux \neq \emptyset$ implies that $x \in Aux$,
- the variables of the set Aux do not occur as tests in conditionals or loops of ρ ,
- $IVar(Q) \cap Aux = \emptyset$.

Let ρ' be the program that can be obtained from ρ by deleting all assignments to variables belonging to the set Aux . Then we have the following rule:

$$\frac{\{P\}\rho\{Q\}}{\{P\}\rho'\{Q\}} \quad (AUX)$$

The rule for auxiliary variables can be explained as follows: To be able to prove some properties of a program ρ' it may be necessary to add a number of extra variables and statements to do some bookkeeping. If these additions satisfy the above conditions, we are sure that they do not influence the flow of control of the program, and therefore they can be deleted after the proof of the enlarged program ρ is completed.

Definition 4.17

Next we have a substitution rule to remove instance variables from preconditions:

$$\frac{\{p^{c_n} \downarrow z_{c_n}\}\rho\{Q\}}{\{(p^{c_n}[l/x]) \downarrow z_{c_n}\}\rho\{Q\}} \quad (S1)$$

provided the instance variable x does not occur in ρ or Q . In practice, this rule is mainly used for auxiliary variables.

Definition 4.18

The following substitution rule removes logical variables from the precondition:

$$\frac{\{p^{c_n} \downarrow z_{c_n}\}\rho\{Q\}}{\{(p^{c_n}[l/z]) \downarrow z_{c_n}\}\rho\{Q\}} \quad (S2)$$

provided the logical variable z does not occur in Q .

Definition 4.19

Finally, the following rule makes explicit the knowledge that in the initial state all instance variables are nil:

$$\frac{\{(p^{c_n} \wedge x \doteq \text{nil}) \downarrow z_{c_n}\} \rho\{Q\}}{\{p^{c_n} \downarrow z_{c_n}\} \rho\{Q\}} \quad (\text{INIT})$$

where $x \in \bigcup_c IVar_c^{c_n}$.

5 An example proof

As an illustration of the proof system we shall now formally prove a property of the program listed at the end of section 2. We want to prove that exactly the primes up to n are generated. This amounts to proving the postcondition

$$\forall i (Prime(i) \wedge i \leq n \leftrightarrow \exists p p.m \doteq i) \quad (5.1)$$

Here i and p are logical variables ranging over integers and objects of class P , respectively. The predicate *Prime* holds for an integer if and only if it is a (positive) prime number.

To establish this postcondition we first introduce an auxiliary variable v (for ‘valid’) of type *Bool* in class P , which indicates that the value stored in the variable b is valid. The variable v is false precisely from the moment that the object sends the value of its b variable to its neighbour until the moment that it receives a new value for the variable b . Now we take a global invariant I which is the conjunction of the following assertions:

- $I_1: \forall p (p.m \neq \text{nil} \rightarrow Prime(p.m))$
- $I_2: \forall p \forall p' (p.m \neq \text{nil} \rightarrow (p'.m \doteq nextpr(p.m) \rightarrow p.l \doteq p'))$
- $I_3: \forall p \forall p' (p.b < p'.b \rightarrow p.m > p'.m)$
- $I_4: \forall p \forall p' (p.m \doteq p'.m \rightarrow p \doteq p')$
- $I_5: \forall g \forall i (Prime(i) \wedge i < g.c \rightarrow \exists p (p.m \doteq i \vee p.b \doteq i))$
- $I_6: \forall g \forall p ((p.m \neq \text{nil} \rightarrow 2 \leq p.m < g.c) \wedge (p.b \neq \text{nil} \rightarrow 2 \leq p.b < g.c))$
- $I_7: \forall p \forall p' (p.m \neq \text{nil} \wedge p'.b \neq \text{nil} \wedge p'.v \rightarrow p.m < p'.b)$
- $I_8: \forall p (p.m \neq \text{nil} \wedge p.b \neq \text{nil} \rightarrow p.m < p.b)$

In the above definition g is a logical variable of type G , so it will always denote the root object. The function *nextpr* applied to an integer gives the next prime number.

The assertion I_2 expresses that successive primes are stored in successive P objects in the chain. Assertion I_3 states that the prime candidates flow through the chain in their

natural order. Assertion I_4 states that each prime number is uniquely represented. Next, assertion I_5 states that all the prime numbers among the candidates sent are represented or being processed. On the other hand assertion I_6 essentially expresses that all the numbers which occur in the chain have been sent by the generator. Finally, I_7 and I_8 says that a candidate is always greater than any prime number already found: I_7 states this globally, but only if the variable b containing the candidate is marked as valid by the flag v , and I_8 expresses it locally.

The easiest way to represent the outcome of the local proof system is by means of a *proof outline*. This consists of an expanded version of the program: the statements concerning auxiliary variables are added, the bracketed sections are indicated by angle brackets, and the assertions that play a role in the local proof are inserted at the proper places, surrounded by braces. The proof outline for our example program is given in figure 1.

Now we have to check the assumptions of the cooperation test. Here we shall deal with one pair of I/O statements and one new-statement. We first treat the following case:

$$\{I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s\}(r, ?m) \parallel (s, !!b; v := \text{false})\{I \wedge \psi_1 \downarrow r \wedge (\neg v) \downarrow s\} \quad (5.2)$$

Here s and r are logical variables of type P denoting the sender and the receiver.

We prove (5.2) using the rule (PAR2). The following nontrivial premisses are needed:

$$\{I \wedge \psi_1 \downarrow r\}(s, v := \text{false})\{I \wedge \psi_1 \downarrow r \wedge (\neg v) \downarrow s\} \quad (5.3)$$

$$\{I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s\}(r, ?m) \parallel (s, !!b)\{I \wedge \psi_1 \downarrow r\} \quad (5.4)$$

In order to prove (5.3), by the rule (IASS) and the consequence rule, we must show that the postcondition, after applying the substitution $[\text{false}/s.v]$, is implied by the precondition. Now it is clear that I_1 – I_6 , I_8 , and $\psi_1 \downarrow r$ are not changed by this substitution, so they are subsumed by the precondition. For the other parts we first calculate as follows:

$$\begin{aligned} ((\neg v) \downarrow s)[\text{false}/s.v] &= (\neg s.v)[\text{false}/s.v] \\ &= \neg \text{if } s \doteq s \text{ then false else } s.v \text{ fi} \\ &\equiv \text{true} \end{aligned}$$

Finally, we observe that $I_7[\text{false}/s.v]$ is the formula

$$\forall p \forall p' (p.m \neq \text{nil} \wedge p'.b \neq \text{nil} \wedge \text{if } p' \doteq s \text{ then false else } p'.v \text{ fi} \rightarrow p.m < p'.b)$$

and this is clearly implied by I_7 .

```

⟨P ← {ψ0 : l ≐ nil ∧ m ≐ nil ∧ b ≐ nil}
      ⟨?m⟩; {ψ1 : l ≐ nil ∧ b ≐ nil}
      if m ≠ nil
      then {l ≐ nil}
          ⟨l := new⟩; {true}
          ⟨?b; v := true⟩;
          {ψ2 : v ∧ (b ≠ nil → ∀i(2 ≤ i < m → i ≠ b))};
          while b ≠ nil
          do {v ∧ ∀i(2 ≤ i < m → i ≠ b)}
              if m ≠ b
              then {ψ3 : v ∧ ∀i(2 ≤ i ≤ m → i ≠ b)}
                  ⟨l!b; v := false⟩
              fi; {¬v}
              ⟨?b; v := true⟩ {ψ2}
          od; {b ≐ nil}
          ⟨l!b⟩ {b ≐ nil}
      fi {b ≐ nil}
  : {true}
    ⟨f := new; c := 2⟩; {2 ≤ c ≤ max(2, n + 1)}
    while c ≤ n
    do {2 ≤ c ≤ n}
        ⟨f!c; c := c + 1⟩
    od; {c ≐ max(2, n + 1)}
    ⟨f!nil⟩
    {c ≐ max(2, n + 1)}

```

Figure 1: The proof outline for our example program

By the rule (SR2) and the axiom (COMM) the proof of the second premiss (5.4) amounts to establishing the truth of the formula

$$I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s \wedge s.l \doteq r \rightarrow (I \wedge \psi_1 \downarrow r)[s.b/r.m].$$

Here we only show the following part of this:

$$I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s \wedge s.l \doteq r \rightarrow I_1[s.b/r.m].$$

Now $I_1[s.b/r.m]$ is the formula

$$\forall p(\text{if } p \doteq r \text{ then } s.b \text{ else } p.m \text{ fi} \neq \text{nil} \rightarrow \text{Prime}(\text{if } p \doteq r \text{ then } s.b \text{ else } p.m \text{ fi})),$$

which is equivalent to

$$\forall p((p \neq r \rightarrow p.m \neq \text{nil} \rightarrow \text{Prime}(p.m)) \wedge (p \doteq r \rightarrow (s.b \neq \text{nil} \rightarrow \text{Prime}(s.b)))).$$

For $p \neq r$ we have that $p.m \neq \text{nil} \rightarrow \text{Prime}(p.m)$ is implied by I_1 . For $p \doteq r$ we have to show that $I \wedge \psi_0 \downarrow r \wedge \psi_3 \downarrow s \wedge s.l \doteq r$ implies $\text{Prime}(s.b)$. Here the main point is that there exist no primes between $s.m$ and $s.b$. For suppose that i is the least prime such that $s.m < i < s.b$. Now by I_5 and I_6 there exists a p' such that $p'.m \doteq i$ or $p'.b \doteq i$. If $p'.m \doteq i$ then it follows by I_2 that $s.l \doteq p'$. But, as $s.l \doteq r$ we have $p' \doteq r$, so $p'.m \doteq \text{nil}$, which contradicts $p'.m \doteq i$. Suppose now that $p'.b \doteq i$. We then have that $p'.b < s.b$, so by I_3 it follows that $p'.m > s.m$. But since $p'.m$ is a prime by I_1 , and i is the least prime greater than $s.m$, it follows that $p'.m \geq i \doteq p'.b$, which contradicts I_8 . Therefore there can be no primes between $s.m$ and $s.b$ and then $\psi_3 \downarrow s$ implies that $s.b$ is a prime number.

As another example of an assumption for the cooperation test, we consider the following new-statement:

$$\{I \wedge (l \doteq \text{nil}) \downarrow z\}(z, l := \text{new})\{I \wedge \psi_0 \downarrow z.l\}.$$

Here z is a logical variable ranging over objects of class P. By the axiom (NEW) the proof of this correctness formula amounts to proving the validity of

$$I \wedge (l \doteq \text{nil}) \downarrow z \rightarrow (I \wedge \psi_0 \downarrow z.l)[z'/z.l][\text{new}/z'],$$

where z' is another new logical variable of type P. We only show that $I \wedge (l \doteq \text{nil}) \downarrow z$ implies $I_2[z'/z.l][\text{new}/z']$. Now $I_2[z'/z.l]$ is the formula

$$\forall p \forall p'(p.m \neq \text{nil} \rightarrow p'.m \doteq \text{nextpr}(p.m) \rightarrow \text{if } p \doteq z \text{ then } z' \text{ else } p.l \text{ fi} \doteq p').$$

This is equivalent to

$$\forall p \forall p'(p.m \neq \text{nil} \rightarrow p'.m \doteq \text{nextpr}(p.m) \rightarrow (p \doteq z \rightarrow z' \doteq p') \wedge (p \neq z \rightarrow p.l \doteq p')).$$

To this formula we apply the substitution $[new/z']$, which gives us

$$\begin{aligned} & \forall p \forall p' (p.m \neq \text{nil} \rightarrow p'.m \doteq \text{nextpr}(p.m) \rightarrow (p \doteq z \rightarrow \text{false}) \wedge (p \neq z \rightarrow p.l \doteq p')) \\ & \wedge \forall p (p.m \neq \text{nil} \rightarrow \text{nil} \doteq \text{nextpr}(p.m) \rightarrow (p \doteq z \rightarrow \text{true}) \wedge (p \neq z \rightarrow \text{false})) \\ & \wedge \forall p' (\text{nil} \neq \text{nil} \rightarrow p'.m \doteq \text{nextpr}(\text{nil}) \rightarrow (\text{false} \rightarrow \text{false}) \wedge (\text{true} \rightarrow \text{nil} \doteq p')) \\ & \wedge (\text{nil} \neq \text{nil} \rightarrow \text{nil} \doteq \text{nextpr}(\text{nil}) \rightarrow (\text{false} \rightarrow \text{true}) \wedge (\text{true} \rightarrow \text{false})) \end{aligned}$$

Note that the first conjunct corresponds to interpreting p and p' as ranging over the old objects. The second conjunct results from interpreting p' as the new object, the third from interpreting p as the new object, and the last from taking both p and p' as the new object. All conjuncts but the first are trivially true, and we can get the first conjunct from I_2 if we can show that $\forall p \forall p' (p.m \neq \text{nil} \rightarrow p'.m \doteq \text{nextpr}(p.m) \rightarrow p \neq z)$. Let $p \neq \text{nil}$, $p' \neq \text{nil}$, $p.m \neq \text{nil}$, $p'.m \doteq \text{nextpr}(p.m)$, and $p \doteq z$. By I_2 we derive that $z.l \doteq p'$. But this contradicts the precondition $z.l \doteq \text{nil}$.

The other parts of the cooperation test can be dealt with in the same way as the above ones. After that the program rule (PR) can be applied, with the following result:

$$\{\text{true}\} \rho' \{I \wedge g.c \doteq \max(2, n+1) \wedge \forall p p.b \doteq \text{nil}\}$$

Here ρ' is the program with the auxiliary variable v , but by applying the auxiliary variable rule (AUX) the same result can also be obtained for the original program ρ . It is easy to see that the above postcondition implies the desired assertion 5.1, so the latter can be obtained by the consequence rule (PCR).

6 Semantics

In this section we define in a formal way the semantics of the programming language and the assertion languages. First, in section 6.1, we deal with the assertion languages on their own. Then, in section 6.2, we give a formal semantics to the programming language, making use of *transition systems*. Finally, section 6.3 formally defines the notion of truth of a correctness formula.

6.1 Semantics of the assertion languages

For every type $a \in C^+$, we shall let O^a denote the set of objects of type a , with typical element α^a . To be precise, we define $O^{\text{Int}} = \mathbb{Z}$ and $O^{\text{Bool}} = \mathbb{B}$, whereas for every class $c \in C$ we just take for O^c an arbitrary infinite set. With O_\perp^d we shall denote $O^d \cup \{\perp\}$, where \perp is a special element not in O^d , which will stand for 'undefined', among others the value of the expression nil . Now for every type $d \in C^+$ we let O^{d*}

denote the set of all finite sequences of elements from \mathbf{O}_\perp^d and we take $\mathbf{O}_\perp^{d^*} = \mathbf{O}^{d^*}$. This means that sequences can contain \perp as a component, but a sequence can never be \perp itself (as an expression of a sequence type, nil just stands for the empty sequence).

Definition 6.1

We shall often use generalized Cartesian products of the form

$$\prod_{i \in A} B(i).$$

As usual, the elements of this set are the functions f with domain A such that $f(i) \in B(i)$ for every $i \in A$.

Definition 6.2

Given a function $f \in A \rightarrow B$, $a \in A$, and $b \in B$, we use the *variant notation* $f\{b/a\}$ to denote the function in $A \rightarrow B$ that satisfies

$$f\{b/a\}(a') = \begin{cases} b & \text{if } a' = a \\ f(a') & \text{otherwise.} \end{cases}$$

Definition 6.3

The set $LState^c$ of *local states* of class c , with typical element θ^c , is defined by

$$LState^c = \mathbf{O}^c \times \prod_d (IVar_d^c \rightarrow \mathbf{O}_\perp^d).$$

A local state θ^c describes in detail the situation of a single object of class c at a certain moment during program execution. The first component determines the identity of the object and the values of the instance variables are given by the second component.

It will turn out to be convenient to define the function $\nabla^c \in \prod_d IVar_d^c \rightarrow \mathbf{O}_\perp^d$ such that $\nabla^c(x) = \perp$, for every $x \in \bigcup_d IVar_d^c$. Note that this function ∇ gives the values of the variables of a newly created object: these are all initialized to nil.

Definition 6.4

The set $GState$ of *global states*, with typical element σ , is defined as follows:

$$GState = \left(\prod_d P^d \right) \times \prod_c \left(\mathbf{O}^c \rightarrow \prod_d (IVar_d^c \rightarrow \mathbf{O}_\perp^d) \right)$$

where P^c , for every $c \in C$, denotes the set of finite subsets of \mathbf{O}^c , and for $d = \text{Int}, \text{Bool}$ we define $P^d = \{\mathbf{O}^d\}$.

A global state describes the situation of a complete system of objects at a certain moment during program execution. The first component specifies for each class the set of *existing* objects of that class, that is, the set of objects that have been created up to this point in the execution of the program. Relative to some global state σ an object $\alpha \in \mathbf{O}^d$ can be said to exist if $\alpha \in \sigma_{(1)(d)}$. For the built-in data types we have for every global state σ that $\sigma_{(1)(\text{Int})} = \mathbf{Z}$ and $\sigma_{(1)(\text{Bool})} = \mathbf{B}$. Note that $\perp \notin \sigma_{(1)(d)}$ for every $d \in C^+$. The second component of a global state specifies for each object the values of its instance variables.

We introduce the following abbreviations: $\sigma_{(1)(d)}$ will be abbreviated to $\sigma^{(d)}$, and $\sigma^{(d)} \cup \{\perp\}$ to $\sigma_{\perp}^{(d)}$. Whenever it is clear from the context that $\alpha \in \mathbf{O}^c$, we abbreviate $\sigma_{(2)(c)}(\alpha)$ by $\sigma(\alpha)$. Furthermore, for any variable $x \in IVar_d^c$, we abbreviate $\sigma_{(2)(c,d)}(\alpha)(x)$, the value of the variable x of the object α , by $\sigma(\alpha)(x)$.

Definition 6.5

We now define the set $LEnv$ of *logical environments*, with typical element ω , by

$$LEnv = \prod_a (LogVar_a \rightarrow \mathbf{O}_{\perp}^a).$$

A logical environment assigns values to logical variables. We abbreviate $\omega_{(a)}(z_a)$ to $\omega(z_a)$.

Definition 6.6

The following semantic functions are defined in a straightforward manner. We omit most of the detail and only give the most important cases:

1. The function $\mathcal{E}_d^c \in Exp_d^c \rightarrow LState^c \rightarrow \mathbf{O}_{\perp}^d$ assigns a value $\mathcal{E}[e](\theta)$ to the expression e_d^c in the local state θ^c .
2. The function $\mathcal{L}_d^c \in LExp_d^c \rightarrow LEnv \rightarrow LState^c \rightarrow \mathbf{O}_{\perp}^d$ assigns a value $\mathcal{L}[I](\omega)(\theta)$ to the local expression I_d^c in the logical environment ω and the local state θ^c .
3. The function $\mathcal{G}_a \in GExp_a \rightarrow LEnv \rightarrow GState \rightarrow \mathbf{O}_{\perp}^a$ assigns a value $\mathcal{G}[g](\omega)(\sigma)$ to the global expression g_a in the logical environment ω and the global state σ .
4. The function $\mathcal{A}^c \in LAss^c \rightarrow LEnv \rightarrow LState^c \rightarrow \mathbf{B}$ assigns a value $\mathcal{A}[p](\omega)(\theta)$ to the local assertion p^c in the logical environment ω and the local state θ^c . Here the following cases are special:

$$\mathcal{A}[I_{\text{Bool}}](\omega)(\theta) = \begin{cases} \text{true} & \text{if } \mathcal{L}[I](\omega)(\theta) = \text{true} \\ \text{false} & \text{if } \mathcal{L}[I](\omega)(\theta) = \text{false or } \mathcal{L}[I](\omega)(\theta) = \perp \end{cases}$$

$$\mathcal{A}[\exists z_d P](\omega)(\theta) = \begin{cases} \text{true} & \text{if there is an } \alpha^d \in \mathbf{O}^d \text{ such that } \mathcal{A}[P](\omega\{\alpha/z\})(\theta) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

Note that in the latter case $d = \text{Int}$ or $d = \text{Bool}$ and that the range of quantification *does not include* \perp .

5. The function $\mathcal{A} \in GAss \rightarrow LEnv \rightarrow GState \rightarrow \mathbf{B}$ assigns a value $\mathcal{A}[P](\omega)(\sigma)$ to the global assertion P in the logical environment ω and the global state σ . The following cases are special:

$$\mathcal{A}[g_{\text{Bool}}](\omega)(\sigma) = \begin{cases} \text{true} & \text{if } \mathcal{L}[g](\omega)(\sigma) = \text{true} \\ \text{false} & \text{if } \mathcal{L}[g](\omega)(\sigma) = \text{false or } \mathcal{L}[g](\omega)(\sigma) = \perp \end{cases}$$

$$\mathcal{A}[\exists z_d P](\omega)(\sigma) = \begin{cases} \text{true} & \text{if there is an } \alpha^d \in \sigma^{(d)} \text{ such that } \mathcal{A}[P](\omega\{\alpha/z\})(\sigma) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

Note that here d can be any type in C^+ and that the quantification ranges over $\sigma^{(d)}$, the set of *existing* objects of type d (which does not include \perp).

$$\mathcal{A}[\exists z_{d^*} P](\omega)(\sigma) = \begin{cases} \text{true} & \text{if there is an } \alpha^{d^*} \in \mathbf{O}^{d^*} \text{ such} \\ & \text{that } \alpha(n) \in \sigma_{\perp}^{(d)} \text{ for all } n \in \mathbf{N} \\ & \text{and } \mathcal{A}[P](\omega\{\alpha/z\})(\sigma) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

For sequence types, quantification ranges over those sequences of which every element is either \perp or an existing object.

The values $\mathcal{G}[g_a](\omega)(\sigma)$ of the global expression g_a and $\mathcal{A}[g](\omega)(\sigma)$ of the global assertion P are in fact only meaningful for those ω and σ that are consistent and compatible:

Definition 6.7

We define the global state σ to be *consistent*, for which we use the notation $OK(\sigma)$ iff

1. $\forall c \in C \forall \alpha \in \sigma^{(c)} \forall d \in C \forall x \in IVar_d^c \sigma(\alpha)(x) \in \sigma_{\perp}^{(d)}$
2. $\forall c \in C \forall \beta \in \mathbf{O}^c \setminus \sigma^{(c)} \forall d \in C^+ \forall x \in IVar_d^c \sigma(\beta)(x) = \perp$

In other words, the value in σ of a variable of an existing object is either \perp or an existing object itself, and the variables of non-existing objects are uninitialized.

Furthermore we define the logical environment ω to be *compatible* with the global state σ , with the notation $OK(\omega, \sigma)$, iff $OK(\sigma)$ and, additionally,

$$\forall d \in C \forall z \in \text{LogVar}_d \omega(z) \in \sigma_{\perp}^{(d)}$$

and

$$\forall d \in C \forall z \in \text{LogVar}_d \forall a \in \mathbf{N} \omega(z)(a) \in \sigma_{\perp}^{(d)}.$$

In other words, ω assigns to every logical variable z_d of a simple type the value \perp or an existing object, and to every sequence variable z_{d^*} a sequence of which each element is an existing object or equals \perp .

6.2 The transition system

We will describe the internal behaviour of an object by means of a transition system. A *local configuration* we define to be a pair (S^c, θ^c) . The set of local configurations is denoted by $LConf$. Let $Rec = \{ \langle \alpha, \beta \rangle, \langle \alpha, \beta, \gamma \rangle : \alpha, \beta \in \bigcup_c \mathbf{O}_{\perp}^c, \gamma \in \bigcup_d \mathbf{O}_{\perp}^d \} \cup \{ \epsilon \}$. A pair $\langle \alpha, \beta \rangle$ is called an *activation record*. It records the information that the object α created β . A triple $\langle \alpha, \beta, \gamma \rangle$ is called a *communication record*. It records the information that the object γ is sent by α to β . ϵ will denote a transition which is due to a local computation. We define for every $r \in Rec$ a transition relation $\rightarrow^r \subseteq LConf \times LConf$. To facilitate the semantics we introduce the auxiliary statement E , the empty statement, to denote termination.

Definition 6.8

We define

- $(x_d := e_d, \theta) \rightarrow^{\epsilon} (E, \theta')$,
let $\theta = \langle \alpha, s \rangle$, then $\theta' = \langle \alpha, s \{ \mathcal{E} \| e_d \| (\theta) / x \} \rangle$.
- $(x_d := \text{new}, \theta) \rightarrow^{\langle \alpha, \beta \rangle} (E, \theta')$,
let $\theta = \langle \alpha, s \rangle$, then $\theta' = \langle \alpha, s \{ \beta / x_d \} \rangle$ and $\beta \in \mathbf{O}^d$.
- $(x_c ! e_d, \theta) \rightarrow^{\langle \alpha, \beta, \gamma \rangle} (E, \theta)$,
let $\theta = \langle \alpha, s \rangle$, then $\beta = s(x) \neq \perp$ and $\gamma = \mathcal{E} \| e_d \| (\theta)$.
- $(?y_d, \theta) \rightarrow^{\langle \perp, \beta, \gamma \rangle} (E, \theta')$,
let $\theta = \langle \beta, s \rangle$, then $\theta' = \langle \beta, s \{ \gamma / y_d \} \rangle$ and $\gamma \in \mathbf{O}_{\perp}^d$.
- $(x_c ? y_d, \theta) \rightarrow^{\langle \alpha, \beta, \gamma \rangle} (E, \theta')$,
let $\theta = \langle \beta, s \rangle$, then $\theta' = \langle \beta, s \{ \gamma / y_d \} \rangle$, $\alpha = s(x_c) \neq \perp$, and $\gamma \in \mathbf{O}_{\perp}^d$.
- $(S, \theta) \rightarrow^{\epsilon} (S, \theta)$.

- $(E; S, \theta) \rightarrow^e (S, \theta).$
- $$\frac{(S_1, \theta) \rightarrow^r (S_2, \theta')}{(S_1; S, \theta) \rightarrow^r (S_2; S, \theta')}$$
- (if e_{Bool} then S_1 else S_2 fi, θ) $\rightarrow^e (S_1, \theta)$,
if $\mathcal{E}\|e_{\text{Bool}}\|(\theta) = \text{true}.$
- (if e_{Bool} then S_1 else S_2 fi, θ) $\rightarrow^e (S_2, \theta)$,
if $\mathcal{E}\|e_{\text{Bool}}\|(\theta) = \text{false}.$
- (while e_{Bool} do S od, θ) $\rightarrow^e (S; \text{while } e_{\text{Bool}} \text{ do } S \text{ od}, \theta)$,
if $\mathcal{E}\|e_{\text{Bool}}\|(\theta) = \text{true}.$
- (while e_{Bool} do S od, θ) $\rightarrow^e (E, \theta)$,
if $\mathcal{E}\|e_{\text{Bool}}\|(\theta) = \text{false}.$

Note that locally the value which is received when executing an input statement is chosen arbitrarily, the same holds for the identity of the object created by the execution of a new-statement. We define $\bigcup_h \rightarrow^h = TC(\bigcup_{r \in Rec} \rightarrow^r)$. Here the operation TC denotes the transitive closure which composes additionally the communication records and activation records into a *history* h , a sequence of communication records and activation records.

Next we describe the behaviour of several objects working in parallel. The local behaviour of the objects we shall derive from the local transition system as described above. But at this level we have the necessary information to select the right choices.

We define an *intermediate configuration* to be a tuple $(\sigma, (\alpha_i, S_i^{c_i})_i)$, where $\alpha_i \in \sigma^{(c_i)}$, assuming all the α_i to be distinct. The set of intermediate configurations will be denoted by $IConf$. We define $\rightarrow^r \subseteq IConf \times IConf$ as follows (note that we use the same notation as for the local transition relation, however this will cause no harm):

Definition 6.9

We define

- $$\frac{(S_j, \theta_1) \rightarrow^e (S'_j, \theta_2)}{(\sigma, (\alpha_i, S_i)_i) \rightarrow^e (\sigma', (\alpha_i, S'_i)_i)}$$

where $\theta_1 = \langle \alpha_j, \sigma(\alpha_j) \rangle$,

$$S'_i = S_i \quad i \neq j$$

$$= S'_j \quad \text{otherwise,}$$
and $\sigma'_{(1)} = \sigma_{(1)}$, $\sigma'_{(2)} = \sigma_{(2)}\{s/\alpha_j\}$, with $\theta_2 = \langle \alpha_j, s \rangle$.

- $$\frac{(S_j, \theta_1) \rightarrow^{<\alpha_j, \beta>} (S'_j, \theta_2)}{(\sigma, (\alpha_i, S_i)_i) \rightarrow^{<\alpha_j, \beta>} (\sigma', (\alpha_i, S'_i)_i)}$$

where $\theta_1 = \langle \alpha_j, \sigma(\alpha_j) \rangle$,
 $S'_i = S_i \quad i \neq j$
 $= S'_i \quad \text{otherwise,}$
and $\beta \in \mathbf{O}^d \setminus \sigma^{(d)}$, $\sigma'_{(1)} = \sigma_{(1)} \{ \sigma^{(d)} \cup \{ \beta \} / d \}$, and $\sigma'_{(2)} = \sigma_{(2)} \{ s / \alpha_j \} \{ \nabla / \beta \}$, with $\theta_2 = \langle \alpha_j, s \rangle$.
- $$\frac{(S_j, \theta_1) \rightarrow^r (S'_j, \theta_2), (S_k, \theta'_1) \rightarrow^{r'} (S'_k, \theta'_2)}{(\sigma, (\alpha_i, S_i)_i) \rightarrow^r (\sigma', (\alpha_i, S'_i)_i)}$$

where $\theta_1 = \langle \alpha_j, \sigma(\alpha_j) \rangle$, $\theta'_1 = \langle \alpha_k, \sigma(\alpha_k) \rangle$, $j \neq k$, and $r = \langle \alpha_j, \alpha_k, \gamma \rangle$, $r' = \langle \alpha'_j, \alpha_k, \gamma \rangle$, with $\gamma \in \bigcup_d \mathbf{O}^d_\perp$ and $\alpha'_j = \alpha_j, \perp$, furthermore we have
 $S'_i = S_i \quad i \neq j, k$
 $= S'_i \quad i = j, k,$
and $\sigma'_{(1)} = \sigma_{(1)}$, $\sigma'_{(2)} = \sigma_{(2)} \{ s_1 / \alpha_j \} \{ s_2 / \alpha_k \}$, with $\theta_2 = \langle \alpha_j, s_1 \rangle$ and $\theta'_2 = \langle \alpha_k, s_2 \rangle$.

The first rule above selects one object and its local state and uses the local transition system to derive one local step of this object. The second rule selects an object which is about to create a new object. The variables of this new object are initialized to nil. The local state of the creator is modified according to the local transition system. Note that knowing the set of existing objects we can now indeed select a new object. The third rule finally selects two objects which are ready to communicate with each other. Note that $r' = \langle \perp, \alpha_k, \gamma \rangle$ models the possibility that α_k executes an input statement of the form $?y$.

We define $\bigcup_h \rightarrow^h = TC(\bigcup_{r \in Rec} \rightarrow^r)$, again using the same notation as for the transitive closure of the local transition relation, from the context however it should be clear which one is meant.

To describe the behaviour of a complete system we introduce the notion of a *global configuration*: a pair (X, σ) , where $X \in \prod_c \mathbf{O}^c \rightarrow Stat^c$, and a transition relation $\rightarrow^r \subseteq GConf \times GConf$. We note again that we do not notationally distinguish between the different transition relations, from the context however it will be clear which one is meant. The set of all global configurations we denote by $GConf$. We will abbreviate in the sequel $X(c)(\alpha)$, for $\alpha \in \mathbf{O}^c$, by $X(\alpha)$. The idea is that $X(\alpha)$ denotes the statement to be executed by α .

Definition 6.10

We have the following rule

$$\frac{(\sigma, (\alpha_i, X(\alpha_i))_i) \rightarrow^r (\sigma', (\alpha_i, S_i^{c_i})_i)}{(X, \sigma) \rightarrow^r (X', \sigma')}$$

where $\alpha_i \in \sigma^{(c_i)}$ (all the α_i distinct) and $X' = X\{S_i^{c_i}/\alpha_i\}_i$.

This rule selects some finite set of objects which execute in parallel according to the previous transition system.

We define $\bigcup_h \rightarrow^h = TC(\bigcup_{r \in Rec} \rightarrow^r)$. We proceed with the following definition which characterizes the set of *initial* and *final* global configurations of a given program ρ :

Definition 6.11

Let $\rho = \langle U : S_n^{c_n} \rangle$, with $U = c_1 \leftarrow S_1^{c_1}, \dots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}}$. Furthermore let $X \in \prod_c \mathbf{O}^c \rightarrow \mathbf{Stat}^c$. We define

$Init_\rho(X)$ iff

- $X(\alpha) = S_i^{c_i}$, $c_i \in \{c_1, \dots, c_n\}$, $\alpha \in \mathbf{O}^{c_i}$.
- $X(\alpha) = E$, $\alpha \in \mathbf{O}^c$, $c \notin \{c_1, \dots, c_n\}$.

We define for a state σ such that $OK(\sigma)$:

$Init_\rho(\sigma)$ iff

- $\sigma^{(c)} = \emptyset$ $c \in \{c_1, \dots, c_{n-1}\}$
 $= \{\alpha\}$ $c = c_n$, for some $\alpha \in \mathbf{O}^{c_n}$
- $\sigma(\alpha)(x) = \perp$, for $\alpha \in \sigma^{(c_n)}$ and $x \in IVar_{c_n}^{c_n}$.

We next define $Init_\rho((X, \sigma))$ iff $Init_\rho(X)$ and $Init_\rho(\sigma)$. Finally, we define

$Final_\rho((X, \sigma))$ iff $X(\alpha) = E$, $c_i \in \{c_1, \dots, c_n\}$, for $\alpha \in \sigma^{(c_i)}$.

The predicate $Init_\rho(Final_\rho)$ characterizes the set of *initial* (*final*) configurations of θ . Note that the value of a variable $x_c^{c_n}$ of the root-object, $c \in \{c_1, \dots, c_n\}$, is undefined initially. This follows for $c \neq c_n$ from the fact that we consider only consistent states and that initially only the root-object exists (with respect to the classes c_1, \dots, c_n). But the consistency of the initial state would also allow the value of a variable $x \in IVar_{c_n}^{c_n}$ to be the root-object itself. However, as it will appear to be convenient with respect to the formulation of some rules which formalize reasoning about the initial state, we define the initial state to be completely specified by the variables ranging over the standard objects.

Now we are able to define the meaning of the following programming constructs: S^c , (z_c, S^c) , $(z_{c_i}, S_1^{c_i}) \parallel (z_{c_j}, S_2^{c_j})$ and ρ .

Definition 6.12

We define

$$S[S^c](\theta_1^c) = \{\theta_2^c : \text{for some } h (S^c, \theta_1^c) \rightarrow^h (E, \theta_2^c)\}$$

Definition 6.13

We define $\mathcal{I}[(z_c, S^c)](\omega)(\sigma_1) = \emptyset$ if not $OK(\omega, \sigma_1)$, and $\mathcal{I}[(z_{c_i}, S_1^{c_i}) \parallel (z_{c_j}, S_2^{c_j})](\omega)(\sigma_1) = \emptyset$ if not $OK(\omega, \sigma_1)$ or $\omega(z_{c_i}) = \omega(z_{c_j})$. So assume from now on that $OK(\omega, \sigma_1)$, furthermore that $\omega(z_c) = \alpha$, $\omega(z_{c_i}) = \alpha_i$, and $\omega(z'_{c_j}) = \alpha'_j$.

$$\mathcal{I}[(z_c, S^c)](\omega)(\sigma_1) = \{\sigma_2 : \text{for some } h (\sigma_1, (\alpha, S^c)) \rightarrow^h (\sigma_2, (\alpha, E))\}$$

Assuming furthermore that $\alpha_i \neq \alpha'_j$:

$$\begin{aligned} \mathcal{I}[(z_{c_i}, S_1^{c_i}) \parallel (z'_{c_j}, S_2^{c_j})](\omega)(\sigma_1) = \\ \{\sigma_2 : \text{for some } h (\sigma_1, (\alpha_i, S_1^{c_i}), (\alpha'_j, S_2^{c_j})) \rightarrow^h (\sigma_2, (\alpha_i, E), (\alpha'_j, E))\} \end{aligned}$$

Definition 6.14

The semantics of programs is defined as follows:

$$\mathcal{P}[\rho](\sigma_1) = \{\sigma_2 : \text{for some } h (X_1, \sigma_1) \rightarrow^h (X_2, \sigma_2)\}$$

where $Init_\theta((X_1, \sigma_1))$ and $Final_\theta((X_2, \sigma_2))$.

Note that $\mathcal{P}[\rho](\sigma) = \emptyset$ if it is not the case that $Init_\rho(\sigma)$.

6.3 Truth of correctness formulas

In this section we define formally the truth of the local, intermediate, and global correctness formulas, respectively. First we define the truth of local correctness formulas.

Definition 6.15

We define

$$\models \{p^c\}S^c\{q^c\} \text{ iff } \forall \omega, \theta_1, \theta_2 \in S[S^c](\theta_1) : \theta_1, \omega \models p^c \Rightarrow \theta_2, \omega \models q^c.$$

Next we define the truth of intermediate correctness formulas.

Definition 6.16

We define

$$\models \{P\}(z_c, S^c)\{Q\} \text{ iff} \\ \forall \omega, \sigma_1, \sigma_2 \in \mathcal{I}[(z_c, S^c)](\omega)(\sigma_1) : \sigma_1, \omega \models P \Rightarrow \sigma_2, \omega \models Q.$$

And

$$\models \{P\}(z_{c_i}, S^{c_i}) \parallel (z'_{c_j}, S^{c_j})\{Q\} \text{ iff} \\ \forall \omega, \sigma_1, \sigma_2 \in \mathcal{I}[(z_{c_i}, S^{c_i}) \parallel (z'_{c_j}, S^{c_j})](\omega)(\sigma_1) : \sigma_1, \omega \models P \Rightarrow \sigma_2, \omega \models Q.$$

Finally, we define the truth of global correctness formulas.

Definition 6.17

We define

$$\models \{P\}\rho\{Q\} \text{ iff } \forall \omega, \sigma_1, \sigma_2 \in \mathcal{P}[\rho](\sigma_1) : \sigma_1, \omega \models P \Rightarrow \sigma_2, \omega \models Q.$$

7 Soundness

In this section we prove the soundness of the proof system as presented in the previous section. The soundness of the local proof system is proved by a straightforward induction on the length of the derivation (see, for example, [Ap1]). In the following subsection we discuss the soundness of the intermediate proof system.

7.1 The intermediate proof system

We prove the soundness of the assignment axiom (IASS) and the axiom (NEW). The soundness of the intermediate proof system then follows by a straightforward induction argument. To prove the soundness of the assignment axiom (IASS) we need the following lemma about the correctness of the corresponding substitution operation. This lemma states that semantically substituting the expression g' for $z.x$ in an assertion (expression) yields the same result when evaluating the assertion (expression) in the state where the value of g' is assigned to the variable x of the object denoted by z .

Lemma 7.1

For an arbitrary σ, ω such that $OK(\omega, \sigma)$ we have:

$$\mathcal{G}\llbracket g[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) = \mathcal{G}\llbracket g \rrbracket(\omega)(\sigma')$$

and

$$\mathcal{A}\llbracket P[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) = \mathcal{A}\llbracket P \rrbracket(\omega)(\sigma')$$

where $\sigma'_{(1)} = \sigma_{(1)}$ and $\sigma'_{(2)} = \sigma_{(2)}\{\mathcal{G}\llbracket g'_d \rrbracket(\omega)(\sigma)/\omega(z_c), x_d\}$.

Proof

By induction on the complexity of g and P . We treat only the case $g = g_1.x$, all the other ones following directly from the induction hypothesis. Now:

$$\begin{aligned} \mathcal{G}\llbracket g[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) &= \\ \mathcal{G}\llbracket \text{if } g_1[g'_d/z_c.x_d] \doteq z_c \text{ then } g' \text{ else } g_1[g'/z_c.x].x \text{ fi} \rrbracket(\omega)(\sigma) \end{aligned}$$

Suppose that $\mathcal{G}\llbracket g_1[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) = \omega(z_c)$. We have: $\mathcal{G}\llbracket g_1.x \rrbracket(\omega)(\sigma') = \sigma'(\mathcal{G}\llbracket g_1 \rrbracket(\omega)(\sigma'))(x)$. So by the induction hypothesis we have that:

$$\mathcal{G}\llbracket g_1.x \rrbracket(\omega)(\sigma') = \sigma'(\omega(z_c))(x) = \mathcal{G}\llbracket g'_d \rrbracket(\omega)(\sigma).$$

On the other hand if $\mathcal{G}\llbracket g_1[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) \neq \omega(z_c)$ then:

$$\begin{aligned} \mathcal{G}\llbracket g_1[g'_d/z_c.x_d].x_d \rrbracket(\omega)(\sigma) &= \\ \sigma(\mathcal{G}\llbracket g_1[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma))(x_d) &= \text{(definition of } \sigma') \\ \sigma'(\mathcal{G}\llbracket g_1[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma))(x_d) &= \text{(induction hypothesis)} \\ \sigma'(\mathcal{G}\llbracket g_1 \rrbracket(\omega)(\sigma'))(x_d) &= \\ \mathcal{G}\llbracket g_1.x_d \rrbracket(\omega)(\sigma'). \end{aligned}$$

□

The following lemma states the soundness of the axiom (IASS)

Lemma 7.2

We have

$$\models \left\{ P[e \downarrow z/z.x] \right\} (z, x := e) \left\{ P \right\},$$

where we assume $x := e \in \text{Stat}^c$ and $z \in \text{LogVar}_c$.

Proof

Let σ, ω , with $OK(\omega, \sigma)$, such that $\sigma, \omega \models P[e \downarrow z/z.x]$ and $\sigma' \in \mathcal{I}[(z, x := e)](\omega)(\sigma)$. It follows that $\sigma'_{(1)} = \sigma_{(1)}$ and $\sigma'_{(2)} = \sigma_{(2)}\{\mathcal{G}\llbracket e \downarrow z \rrbracket(\omega)(\sigma)/\omega(z), x\}$ (note that $\mathcal{G}\llbracket e \downarrow$

$z](\omega)(\sigma) = \mathcal{E}[e](\langle \alpha, \sigma(\alpha) \rangle)$, with $\alpha = \omega(z)$. Thus by the previous lemma we conclude $\sigma', \omega \models P$. \square

To prove the soundness of the axiom describing the new statement we need the following lemma which states the correctness of the corresponding substitution operation. This lemma states that semantically the substitution $[new/z]$ applied to an assertion yields the same result when evaluating the assertion in the state resulting from the creation of a new object, interpreting the variable z as the newly created object.

Lemma 7.3

For an arbitrary $\omega, \omega', \sigma, \sigma', \beta \in \mathbf{O}^c \setminus \sigma^{(c)}$ such that $OK(\omega, \sigma)$ and

$$\begin{aligned}\sigma'_{(1)} &= \sigma_{(1)}\{\sigma^{(c)} \cup \{\beta\}/c\} \\ \sigma'_{(2)} &= \sigma_{(2)}\{\nabla/\beta\} \\ \omega' &= \omega\{\beta/z_c\},\end{aligned}$$

we have for an arbitrary assertion P :

$$\mathcal{A}[P[new/z_c]](\omega)(\sigma) = \mathcal{A}[P](\omega')(\sigma').$$

The proof of this lemma proceeds by induction on the structure of P . To carry out this induction argument, which we trust the interested reader to be able to perform, we need the following two lemmas. The first of which is applied to the case $P = g$ and the second of which is applied to the case $P = \exists zP', z \in LVar_a, a = c, c^*$.

Lemma 7.4

For an arbitrary σ, ω , with $OK(\omega, \sigma)$, global expression g and logical variable z_c such that $g[new/z_c]$ is defined we have:

$$\mathcal{G}[g](\omega')(\sigma') = \mathcal{G}[g[new/z_c]](\omega)(\sigma)$$

where $\sigma'_{(1)} = \sigma_{(1)}\{\sigma^{(c)} \cup \{\beta\}/c\}$, $\sigma'_{(2)} = \sigma_{(2)}\{\nabla/\beta\}$, and $\omega' = \omega\{\beta/z_c\}$, $\beta \notin \sigma^{(c)}$.

Proof

Induction on the structure of g . \square

The following lemma states that semantically the substitution $[z_{\text{Bool}}, z_c/z_c]$ applied to an assertion (expression) yields the same result when updating the sequence denoted by the variable z_c to the value of z_c at those positions for which the sequence denoted by z_{Bool} gives the value true.

Lemma 7.5

Let $\omega, \sigma, \alpha = \omega(z_c^\bullet), \alpha' = \omega(z_{\text{Bool}^\bullet})$ such that $|\alpha| = |\alpha'|$ and $OK(\omega, \sigma)$.

Let $\alpha'' \in \mathbf{O}^{c^\bullet}$ such that

- $|\alpha''| = |\alpha|$
- for $n \in \mathbf{N}$: $\alpha''(n) = \omega(z_c) \text{ if } \alpha'(n) = \text{true}$
 $= \alpha(n) \text{ if } \alpha'(n) = \text{false}$
 $= \perp \text{ if } \alpha'(n) = \perp$

Let $\omega' = \omega\{\alpha''/z_c^\bullet\}$. Then:

1. For every g such that $g[z_{\text{Bool}^\bullet}, z_c/z_c^\bullet]$ is defined:

$$\mathcal{G}\|g[z_{\text{Bool}^\bullet}, z_c/z_c^\bullet]\|(\omega)(\sigma) = \mathcal{G}\|g\|(\omega')(\sigma)$$

2. For every P such that z_{Bool^\bullet} does not occur in it:

$$\mathcal{A}\|P[z_{\text{Bool}^\bullet}, z_c/z_c^\bullet]\|(\omega)(\sigma) = \mathcal{A}\|P\|(\omega')(\sigma)$$

Proof

Induction on the structure of g and P . □

Now we are ready to prove the soundness of the axiom (NEW).

Lemma 7.6

We have

$$\models \left\{ P[z'/z.x][\text{new}/z'] \right\} (z, x := \text{new}) \left\{ P \right\},$$

where $x := \text{new} \in \text{Stat}^c$, $z \in \text{LogVar}_c$, and z' is a new logical variable of the same type as x .

Proof

Let σ, ω , with $OK(\sigma, \omega)$, such that $\sigma, \omega \models P[z'/z.x][\text{new}/z']$ and $\sigma' \in \mathcal{I}[(z, x := \text{new})](\omega)(\sigma)$. We have by lemma 7.3 that $\sigma'', \omega' \models P[z'/z.x]$, where $\omega' = \omega\{\beta/z'\}$, with $\beta \in \mathbf{O}^d \setminus \sigma^d$, assuming d to be the type of the variable x , and $\sigma''_{(1)} = \sigma_{(1)}\{\sigma^{(d)} \cup \{\beta\}/d\}$, $\sigma''_{(2)} = \sigma_{(2)}\{\nabla/\beta\}$. Now by lemma 7.1 it follows that $\sigma', \omega' \models P$. Finally, as z' does not occur in P we have $\sigma', \omega \models P$. □

7.2 The global proof system

In this subsection we prove the soundness of the global proof system. We will prove only the soundness of the rule (PR), the other rules being straightforward to deal with. The problem with proving the soundness of the rule (PR) is how to interpret the premise $A \vdash \{p\}S\{q\}$. We solve this problem by showing that a proof $A \vdash \{p\}S\{q\}$ essentially boils down to a finite conjunction of local assertions. We start with the following two definitions.

Definition 7.7

Given a bracketed program ρ , R a substatement of ρ , we define R to be *normal* iff every bracketed section of U occurs inside or outside of R .

Next we define $After(R, S)$, where R is a substatement of S , to be the statement to be executed when the execution of R has just terminated. On the other hand we will define $Before(R, S)$ to be the statement to be executed when the execution of R is about to start.

Definition 7.8

Let R be a substatement of the statement S . We define $After(R, S)$ as follows:

- If $R = S$ then $After(R, S) = E$
- If $S = S_1; S_2$
 then $After(R, S) = After(R, S_1); S_2$ if R occurs in S_1
 $After(R, S) = After(R, S_2)$ otherwise
- If $S = \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}$
 then $After(R, S) = After(R, S_1)$ if R occurs in S_1
 $After(R, S) = After(R, S_2)$ otherwise
- If $S = \text{while } e \text{ do } S_1 \text{ od then } After(R, S) = After(R, S_1); S$

Next we define $Before(R, S)$ as follows: $Before(R, S) = R; S'$, where $After(R, S) = E; S'$.

Note that for the above definition to be formally correct we have to assume some mechanism which enables one to distinguish between different occurrences of an arbitrary statement. We will simply assume such a mechanism to exist. So in the sequel when referring to a statement we in fact will sometimes mean a particular occurrence of that statement.

We have the following lemma which can be seen as a particular formulation of the soundness of the local proof system.

Lemma 7.9

Let $S \in \text{Stat}^c$ be a statement such that every I/O statement and new statement occurring in it is contained in a bracketed section. Furthermore let A be a set of assumptions about the bracketed sections occurring in S . Then: $A \vdash \{p\}S\{q\}$ iff there exist for every normal substatement R of S local assertions $\text{Pre}(R)$ and $\text{Post}(R)$ such that:

- $\{\text{Pre}(R)\}R\{\text{Post}(R)\} \in A$, for R a bracketed section
- $p \rightarrow \text{Pre}(S), \text{Post}(S) \rightarrow q$
- $\text{Pre}(R) \rightarrow \text{Post}(R)[e/x], R = x := e$
- $\text{Pre}(R) \rightarrow \text{Pre}(R_1), \text{Post}(R_1) \rightarrow \text{Pre}(R_2)$, and $\text{Post}(R_2) \rightarrow \text{Post}(R), R = R_1; R_2$
- $\text{Pre}(R) \wedge e \rightarrow \text{Pre}(R_1), \text{Pre}(R) \wedge \neg e \rightarrow \text{Pre}(R_2), \text{Post}(R_1) \rightarrow \text{Post}(R)$, and $\text{Post}(R_2) \rightarrow \text{Post}(R), R = \text{if } e \text{ then } R_1 \text{ else } R_2 \text{ fi}$
- $\text{Pre}(R) \wedge e \rightarrow \text{Pre}(R_1), \text{Post}(R_1) \rightarrow \text{Pre}(R)$, and $\text{Pre}(R) \wedge \neg e \rightarrow \text{Post}(R), R = \text{while } e \text{ do } R_1 \text{ od}$

Proof

Straightforward induction on the structure of S . □

Given a derivation $A \vdash \{p\}S\{q\}$ we define $VC_A(\{p\}S\{q\})$ to be the set of local assertions corresponding to the last five clauses of the above mentioned lemma.

Given a bracketed program $\rho = \langle U : S_n^{c_n} \rangle$, with $U = c_1 \leftarrow S_1^{c_1}, \dots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}}$, a global assertion I , and the derivations $A_k \vdash \{p_k^{c_k}\}S_k^{c_k}\{q_k^{c_k}\}$, the set of formulas and assertions which have to be verified in the cooperation test will be denoted by $CP(A_1, \dots, A_n, I, z_{c_n})$.

Now we can phrase the soundness of the rule (PR) as follows: If all the local assertions of $VC_{A_k}(\{p_k^{c_k}\}S_k^{c_k}\{q_k^{c_k}\})$, $1 \leq k \leq n$, all the formulas and assertions of $CP(A_1, \dots, A_n, I, z_{c_n})$ are true, and finally the global assertion I satisfies the syntactic restriction mentioned in the cooperation test then the conclusion of the rule (PR) is valid.

It is easy to see that this formulation of the soundness of the rule (PR) is implied by the following theorem.

Theorem 7.10

Let $\rho = \langle U : S_n^{c_n} \rangle$, with $U = c_1 \leftarrow S_1^{c_1}, \dots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}}$, be bracketed.

Suppose that there exist for every normal statement R of ρ local assertions $Pre(R)$ and $Post(R)$ such that for $A_k = \{\{Pre(R)\}R\{Post(R)\} : R \text{ a bracketed section occurring in } S^{c_k}\}$ all the local assertions of $VC_{A_k}(\{p_k^{c_k}\}S_k^{c_k}\{q_k^{c_k}\})$ are true. Assume furthermore that we are given that all the formulas and assertions of $CP(A_1, \dots, A_n, I, z_{c_n})$ are true, and finally that I satisfies the syntactic restriction mentioned in the first clause of the cooperation test.

Let $\tau = (X_i, \sigma_i)_{i=0, \dots, m}$ be a sequence of global configurations such that for every $0 \leq j < m$ we have $(X_j, \sigma_j) \rightarrow^{r_j} (X_{j+1}, \sigma_{j+1})$, where $INIT_\rho((X_0, \sigma_0))$, and for every $c \in \{c_1, \dots, c_n\}$, $\alpha \in \sigma_m^{(c)}$ we have $X_m(\alpha) = E$, or $X_m(\alpha) = Before(R, S^c)$, $After(R, S^c)$, R a bracketed section. Additionally we have to impose the following restriction on the computation τ : For an arbitrary c , $\alpha \in \sigma_m^{(c)}$ we have if $X_i(\alpha) = Before(R, S^c)$, for some $0 \leq i < m$, then for some $i < j < m$ we have $X_j(\alpha) = After(R, S^c)$. This additional condition is necessary to ensure that in the configuration (X_m, σ_m) every existing object is about to enter a bracketed section or has just finished executing one or has finished executing its local process. We have to exclude situations like: $X_m(\alpha) = Before(R, S^c)$, with $R = \text{while } e \text{ do } R_1 \text{ od}; R'$, and $X_{m-1}(\alpha) = After(R_1, S^c)$. Note that in the configuration (X_m, σ_m) the object α is in fact executing *inside* a bracketed section.

Finally, suppose that: $\sigma_0, \omega \models p_n^{c_n} \downarrow z_{c_n}$. Then:

1. $\sigma_m, \omega \models I$
2. For $c_i \in \{c_1, \dots, c_n\}$, $\alpha \in \sigma_m^{(c_i)}$:
 - If $X_m(\alpha) = Before(R, S_i^{c_i})$ then $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models Pre(R)$.
 - If $X_m(\alpha) = After(R, S_i^{c_i})$, and for some $1 \leq k < m$, $X_k(\alpha) = Before(R, S_i^{c_i})$ such that for every $k < l < m$, $X_l(\alpha) = Before(R', S_i^{c_i})$ implies that R' is a proper normal substatement of R , then $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models Post(R)$.
 - If $X_m(\alpha) = E$ then $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models q_i^{c_i}$.

The additional condition in this second clause above can be shown to be necessary as follows: Suppose the statement *if* t *then* R_1 *else* R_2 *fi* occurs in S^c . We then have that $After(R_1, S) = After(R_2, S)$. So we need some additional information about which of the statements R_1, R_2 has actually been executed.

Proof

The proof proceeds by induction on $|h|$, the length of the history of the computation τ .

$|h| = 0$:

Because $\sigma_0, \omega \models p_n^{c_n} \downarrow z_{c_n}$, $INIT_\rho((X_0, \sigma_0))$, and the implication of the last clause of the cooperation test holds, we have that $\sigma_0, \omega \models I$. Now σ_m agrees with σ_0 with respect to the variables occurring in I (the computation τ consists solely of a local computation of the root-object), so $\sigma_m, \omega \models I$. Next we will prove by induction on m , the length of the computation τ (let $\omega(z_{c_n}) = \alpha$):

- If $X_m(\alpha) = \text{Before}(R, S_n^{c_n})$, and R is minimal with respect to the relation “is a proper normal substatement of”, then $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Pre}(R)$.
- If $X_m(\alpha) = \text{After}(R, S_n^{c_n})$, and for some $1 \leq k < m$, $X_k(\alpha) = \text{Before}(R, S_n^{c_n})$, such that for every $k < l < m$, $X_l(\alpha) = \text{Before}(R', S_n^{c_n})$ implies that R' is a substatement of R (this property of R will be abbreviated to “ $\text{After}(R)$ is τ -reachable”), then $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Post}(R)$.
- If $X_m(\alpha) = E$ then $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models q_n^{c_n}$.

The additional condition in the first clause above can be shown to be necessary as follows: Let $R = R_1; R_2$ occur in $S_n^{c_n}$, where $R_1 = \text{while } 0 \leq x \text{ do } x := x - 1 \text{ od}$. Note that $\text{Before}(R, S_n^{c_n}) = \text{Before}(R_1, S_n^{c_n})$. Suppose that $\text{Pre}(R) = 0 < x$ and $\text{Pre}(R_1) = 0 \leq x$. We have $\models 0 < x \rightarrow 0 \leq x$, but of course not the other way around, so when $X_m(\alpha) = \text{Before}(R, S_n^{c_n})$ then $0 < x$ does not hold necessarily. Here we go.

$m = 0$: $X_0(\alpha) = S_n^{c_n}$. We are given that $\models p_n^{c_n} \rightarrow \text{Pre}(S_n^{c_n})$ and $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models p_n^{c_n}$. So we have that $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Pre}(S_n^{c_n})$. Now let R be minimal such that $\text{Before}(R, S_n^{c_n}) = S_n^{c_n}$. Then $S_n^{c_n} = R$ or $S_n^{c_n} = R; R'$, for some R' . In the latter case we are given that $\models \text{Pre}(S_n^{c_n}) \rightarrow \text{Pre}(R)$. So $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Pre}(R)$.

$m > 0$: We distinguish the following cases:

1. $X_m(\alpha) = E; R, E$.

(a) $X_{m-1}(\alpha) = \text{Before}(x := e, S_n^{c_n})$:

By the induction hypothesis we have $\langle \alpha, \sigma_{m-1}(\alpha) \rangle, \omega \models \text{Pre}(x := e)$. We are given that $\models \text{Pre}(x := e) \rightarrow \text{Post}(x := e)[e/x]$. Furthermore we have that $\langle \alpha, \sigma_m(\alpha) \rangle \in \mathcal{S}[x := e](\langle \alpha, \sigma_{m-1}(\alpha) \rangle)$. From which by the soundness of the local assignment axiom we infer $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Post}(x := e)$. Let $\text{After}(R', S_n^{c_n}) = E; R, E$, such that $\text{After}(R')$ is τ -reachable. An easy induction on the complexity of R' establishes that $\models \text{Post}(x := e) \rightarrow \text{Post}(R')$, implying that for $X_m(\alpha) = E$ we have that $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models q_n^{c_n}$ because we are given that $\models \text{Post}(S_n^{c_n}) \rightarrow q_n^{c_n}$.

- (b) $X_{m-1}(\alpha) = \text{Before}(R', S_n^{c_n})$, $R' = \text{while } e \text{ do } R_1 \text{ od}$, and $\langle \alpha, \sigma_{m-1}(\alpha) \rangle \models \neg e$:

By the induction hypothesis we have that $\langle \alpha, \sigma_{m-1}(\alpha) \rangle, \omega \models \text{Pre}(R')$. We are given that $\models \text{Pre}(R') \wedge \neg e \rightarrow \text{Post}(R')$. Furthermore we have that $\sigma_{m-1} = \sigma_m$. Thus $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Post}(R')$. For R'' such that $\text{After}(R'', S_n^{c_n}) = E; R, E$, and $\text{After}(R'')$ is τ -reachable, one can prove by induction on the complexity of R'' that $\models \text{Post}(R') \rightarrow \text{Post}(R'')$. Furthermore, as in the previous case, we have for $X_m(\alpha) = E$ that $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models q_n^{c_n}$.

2. $X_m(\alpha) = R(\neq E)$:

- (a) $X_{m-1}(\alpha) = \text{Before}(R', S_n^{c_n})$, $R' = \text{if } e \text{ then } R_1 \text{ else } R_2 \text{ fi}$, and $\text{Before}(R_1, S_n^{c_n}) = R$:

By the induction hypothesis we have that $\langle \alpha, \sigma_{m-1}(\alpha) \rangle, \omega \models \text{Pre}(R')$. Furthermore we know that $\langle \alpha, \sigma_{m-1}(\alpha) \rangle \models e$. We are given that $\models \text{Pre}(R') \wedge e \rightarrow \text{Pre}(R_1)$. So $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Pre}(R_1)$. Let R'' be minimal such that $\text{Before}(R'', S_n^{c_n}) = R$, it follows that $R_1 = R''$, or $R_1 = R''; R'''$, for some R''' . In the latter case we are given that $\text{Pre}(R_1) \rightarrow \text{Pre}(R'')$, so $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Pre}(R'')$. The case that $R = R_2$ is treated in the same way.

- (b) $X_{m-1}(\alpha) = \text{Before}(R', S_n^{c_n})$, $R' = \text{while } e \text{ do } R_1 \text{ od}$, and $\text{Before}(R_1, S_n^{c_n}) = R$: Analogously to the previous case.

- (c) $X_{m-1}(\alpha) = E; R$:

Let R_1 be the minimal normal statement such that $R = \text{Before}(R_1, S_n^{c_n})$, and R_2 be maximal such that $\text{After}(R_2, S_n^{c_n}) = E; R$, $\text{After}(R_2)$ being τ -reachable. Note that by the additional restriction on the computation τ , which ensures that in the configuration (X_m, σ_m) every existing object is executing outside a bracketed section, we have that R_2 is a *normal* statement. By the induction hypothesis we have that $\langle \alpha, \sigma_{m-1}(\alpha) \rangle, \omega \models \text{Post}(R_2)$. There are the following two cases to consider:

- i. $R_2; R_1$ is a substatement of $S_n^{c_n}$:

We are given that $\models \text{Post}(R_2) \rightarrow \text{Pre}(R_1)$. So we have that $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Pre}(R_1)$.

- ii. $R_1 = \text{while } e \text{ do } R_2 \text{ od}$:

We have that $\models \text{Post}(R_2) \rightarrow \text{Pre}(R_1)$, from which follows the desired result.

Next we consider the case $|h| > 0$:

$h = h_1 \circ < \alpha, \beta >$, where, say, $\alpha \in O^{c_i}, \beta \in O^{c_j}$:

It is not difficult to see that we may assume that

$$\tau = (X_0, \sigma_0), \dots, (X_k, \sigma_k), \dots, (X_l, \sigma_l), \dots, (X_{l'}, \sigma_{l'}), \dots, (X_m, \sigma_m)$$

where,

1. $X_k(\alpha) = \text{Before}(R, S_i^{c_i})$, $X_l(\alpha) = \text{After}(R, S_i^{c_i})$, R the bracketed section execution of which by α consists of the creation of β
2. From (X_k, σ_k) to $(X_{l'}, \sigma_{l'})$ only α is performing
3. From $(X_{l'}, \sigma_{l'})$ to (X_m, σ_m) only β is performing

Of course this can be proved formally. However a formal proof being straightforward but rather tedious notationally we think we are justified in giving only the following informal explanation: Consider the moment of the computation τ that the object α is about to enter the bracketed section execution of which consists of the creation of β . From that moment on all objects execute independently from each other, which implies that from here on we can sequentialize the local computations in an arbitrary way.

Let for $c_k \in \{c_1, \dots, c_n\}$, $\gamma (\neq \alpha, \beta) \in O^{c_k}$, $X_k(\gamma) = X_m(\gamma) = \text{Before}(R', S_k^{c_k})$, R' a bracketed section. By the induction hypothesis we have that $\langle \gamma, \sigma_k(\gamma) \rangle, \omega \models \text{Pre}(R')$. But $\sigma_k(\gamma) = \sigma_m(\gamma)$, so $\langle \gamma, \sigma_m(\gamma) \rangle, \omega \models \text{Pre}(R')$. Analogously for $X_k(\gamma) = X_m(\gamma) = \text{After}(R', S_k^{c_k})$, $E(R'$ a bracketed section).

Furthermore it follows by the induction hypothesis that

$$\sigma_k, \omega \{ \alpha / z_{c_i} \} \models I \wedge \text{Pre}(R) \downarrow z_{c_i},$$

where z_{c_i} does not occur in I . We are given that

$$\models \{ I \wedge \text{Pre}(R) \downarrow z_{c_i} \} (z_{c_i}, R) \{ I \wedge \text{Post}(R) \downarrow z_{c_i} \wedge p^{c_j} \downarrow z_{c_j} [z_{c_i}.x / z_{c_j}] \},$$

assuming that $x := \text{new}$ occurs in R . We have that

$$\sigma_l \in \mathcal{I}[(z_{c_i}, R)](\omega \{ \alpha / z_{c_i} \})(\sigma_k).$$

So we may infer that

$$\sigma_l, \omega \{ \alpha / z_{c_i} \} \models (I \wedge \text{Post}(R) \downarrow z_{c_i} \wedge p^{c_j} \downarrow z_{c_j} [z_{c_i}.x / z_{c_j}]).$$

As z_{c_i} does not occur in I and σ_l agrees with σ_m with respect to the variables occurring in I we have that $\sigma_m, \omega \models I$.

From $\langle \alpha, \sigma_l(\alpha) \rangle, \omega \models \text{Post}(R)$ we infer by an argument similar to the one applied to the case $|h| = 0$ that $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Pre}(R')$, $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models \text{Post}(R')$, $\langle \alpha, \sigma_m(\alpha) \rangle, \omega \models$

$q_i^{c_i}$, in case $X_m(\alpha) = \text{Before}(R', S_i^{c_i})$, $X_m(\alpha) = \text{After}(R', S_i^{c_i})$, R' a bracketed section, and $X_m(\alpha) = E$, respectively. From $\langle \beta, \sigma_l(\beta) \rangle, \omega \models p_j^{c_j}$ we derive in a similar way an analogous result for β . (Note that by the definition of a bracketed section we have $\sigma_l(\alpha)(x) = \beta$.)

Finally, let $h = h_1 \circ \langle \alpha, \beta, \gamma \rangle$, where, say, $\alpha \in O^{c_i}, \beta \in O^{c_j}$:

Again it is not difficult to see that we may assume that

$$\tau = (X_0, \sigma_0), \dots, (X_k, \sigma_k), \dots, (X_l, \sigma_l), \dots, (X_{l'}, \sigma_{l'}), \dots, (X_m, \sigma_m),$$

where

1. $X_k(\alpha) = \text{Before}(R, S_i^{c_i})$, $X_k(\beta) = \text{Before}(R', S_j^{c_j})$, $X_l(\alpha) = \text{After}(R, S_i^{c_i})$, and $X_l(\beta) = \text{After}(R', S_j^{c_j})$, where R, R' are the bracketed sections execution of which by α and β consists of the transfer of γ from α to β
2. From (X_k, σ_k) to (X_m, σ_m) only α and β are performing
3. From (X_l, σ_l) to $(X_{l'}, \sigma_{l'})$ only α is performing
4. From $(X_{l'}, \sigma_{l'})$ to (X_m, σ_m) only β is performing

For $\gamma' \neq \alpha, \beta$ the desired result follows from the induction hypothesis as in the previous case. Furthermore we have by the induction hypothesis that

$$\begin{aligned} \sigma_k, \omega &\models I, \\ \langle \alpha, \sigma_k(\alpha) \rangle, \omega &\models \text{Pre}(R), \text{ and} \\ \langle \beta, \sigma_k(\beta) \rangle, \omega &\models \text{Pre}(R'). \end{aligned}$$

Let $z \in LVar_{c_i}, z' \in LVar_{c_j}$ do not occur in I . It then follows that

$$\sigma_k, \omega \{ \alpha, \beta / z, z' \} \models (I \wedge \text{Pre}(R) \downarrow z \wedge \text{Pre}(R') \downarrow z').$$

Furthermore we are given the truth of the formula

$$\{ I \wedge \text{Pre}(R) \downarrow z \wedge \text{Pre}(R') \downarrow z' \} (z, R) \parallel (z', R') \{ I \wedge \text{Post}(R) \downarrow z \wedge \text{Post}(R') \downarrow z' \},$$

and

$$\sigma_l \in \mathcal{I}[(z_{c_i}, R) \parallel (z'_{c_j}, R')](\omega \{ \alpha, \beta / z_{c_i}, z'_{c_j} \})(\sigma_k).$$

From which we derive that $\sigma_l, \omega \models I$ (so $\sigma_m, \omega \models I$), $\langle \alpha, \sigma_l(\alpha) \rangle, \omega \models \text{Post}(R)$, and $\langle \beta, \sigma_l(\beta) \rangle, \omega \models \text{Post}(R')$. Finally, reasoning in a similar way as in the case $|h| = 0$ will give us the desired result. \square

8 Completeness

In this section we prove the completeness of the proof system, that is, we prove that an arbitrary valid global correctness formula is derivable. Without loss of generality we may assume the sets C and $IVar$ to be finite.

8.1 Histories

First we want to modify a program p by adding to it assignments to so-called *history* variables, i.e., auxiliary variables which record for every object its history, the sequence of communication records and activation records the object participates in. In languages like CSP such histories can be coded by integers: In CSP we can associate with each process an unique integer and thus code a communication record by an integer [Ap2]. As there is no dynamic process creation in CSP a history is a sequence of communication records, which, given the coding of these records, can be coded too.

Given some coding of objects, it is not possible in our language to program an internal computation, using auxiliary variables, which computes the code of an object. That is, we cannot program a mapping of histories into integers. Therefore to be able to prove completeness, using the technique applied to the proof theory of CSP, we have to extend our programming language. We do so by introducing for each $d \in C^+$ a new finite set of instance variables $IVar_d^c$, for an arbitrary c . We define now the set of expressions Exp_a^c , with typical element e_a^c , as follows:

Definition 8.1

$$\begin{array}{lll}
 e_a^c & ::= & x_a^c \\
 & | & \text{self} \quad a = c \\
 & | & \text{nil} \\
 & | & n \quad a = \text{Int} \\
 & | & \text{true} \mid \text{false} \quad a = \text{Bool} \\
 & | & e_{1|\text{Int}}^c + e_{2|\text{Int}}^c \quad e = \text{Int} \\
 & & \vdots \\
 & | & |e_{d^*}^c| \quad a = \text{Int} \\
 & | & \langle e_d^c \rangle \quad a = d^* \\
 & | & e_{1a}^c \circ e_{2a}^c \quad a = d^* \\
 & | & e_{1d}^c \dot{=} e_{2d}^c \quad a = \text{Bool}
 \end{array}$$

The value of the expression $|e_d^c|$ is the length of the sequence denoted by the expression e_d^c . The value of the expression $e_{1a}^c \circ e_{2a}^c$ is the result of concatenating the two sequences denoted by the subexpressions e_{1a}^c, e_{2a}^c . Finally, the expression $\langle e_d^c \rangle$ denotes the sequence consisting of one element, i.e., the value of the expression e_d^c .

The set of statements $Stat^c$ is defined as before, but for the assignment statement. Assignment statements now have the form: $x_a^c := e_a^c$.

Programs are defined as before. Note that we introduced only some very restricted repertoire of operations on sequences in our programming language. The main reason being that more elaborate operations will complicate our substitution operations. However to prove completeness the operations introduced above suffice.

The set of local expressions $LExp_a^c$ is extended as follows:

Definition 8.2

$l_a^c ::=$	z_a	$a = d, d^*, d \in \{\text{Int}, \text{Bool}\}$
	x_a^c	
	self	$a = c$
	nil	
	n	$a = \text{Int}$
	true false	$a = \text{Bool}$
	$l_{1d^*}^c : l_{2\text{Int}}$	$a = d$
	$ l_d^c $	$a = \text{Int}$
	$\langle l_d^c \rangle$	$a = d^*$
	$l_{1a}^c \circ l_{2a}^c$	$e = d^*$
	$l_{1\text{Int}}^c + l_{2\text{Int}}^c$	$a = \text{Int}$
	\vdots	
	$l_{1d}^c \doteq l_{2d}^c$	$a = \text{Bool}$

The added operations on sequences are interpreted as described above. Local assertions are defined as before.

As we do not want to redefine our substitution operations we do *not* change our global assertion language but for allowing instance variables of type d^* , for $d \in C^+$. As a consequence we have to redefine the definition of the transformation of a local

expression, local assertion to a global expression, global assertion, respectively. We do so by viewing the global expressions $\langle g \rangle$, $g_1 \circ g_2$ as abbreviations in the following sense: Suppose the expression $\langle g \rangle$ occurs in the assertion P . Let P' be such that $P'[\langle g \rangle/z] = P$. Then we can view P as an abbreviation of the assertion

$$\exists z (|z| \doteq 1 \wedge z : 1 \doteq g \wedge P').$$

In case an expression of the form $g_1 \circ g_2$ occurs in P , let P' now be such that $P'[g_1 \circ g_2/z] = P$. Then we can view P as an abbreviation of the assertion

$$\begin{aligned} \exists z \quad (& \\ & |z| \doteq |g_1| + |g_2| \wedge \\ & \forall i (1 \leq i \leq |g_1| \rightarrow z : i \doteq g_1 : i) \wedge \\ & \forall i (|g_1| < i \leq |z| \rightarrow z : i \doteq g_2 : i) \wedge \\ & P' \\ &) \end{aligned}$$

Definition 8.3

We can define now $l \downarrow g$ simply as follows:

$$\begin{aligned} z \downarrow g &= z \\ x \downarrow g &= g.x \\ \text{self} \downarrow g &= g \\ \langle l \rangle \downarrow g &= \langle l \downarrow g \rangle \\ l_1 \circ l_2 \downarrow g &= l_1 \downarrow g \circ l_2 \downarrow g \\ &\dots \end{aligned}$$

The transformation $p \downarrow g$ is defined accordingly.

Note that the resulting assertion, in the case that p contains these newly introduced expressions, really is an abbreviation of an assertion.

Next we describe how to code histories in the extended version of our programming language. First we order the set $\{c_1, \dots, c_n, \text{Int}, \text{Bool}\}$, assuming $C = \{c_1, \dots, c_n\}$, as follows:

$$\begin{aligned} \text{Ord}(c_i) &= i \\ \text{Ord}(\text{Int}) &= n + 1 \\ \text{Ord}(\text{Bool}) &= n + 2 \end{aligned}$$

Given this ordering we assume the set of pairs $\langle c, d \rangle$ to be ordered lexicographically. Now we introduce variables which will be used to code the history of an arbitrary object.

Definition 8.4

For an arbitrary c we associate with each pair $\langle c_i, d \rangle$ the variables:

$$\begin{aligned} in1_m &\in IVar_{c_i}^c, \quad in2_m \in IVar_{d^*}^c, \quad in3_m \in IVar_{Int^*}^c, \\ out1_m &\in IVar_{c_i}^c, \quad out2_m \in IVar_{d^*}^c, \quad out3_m \in IVar_{Int^*}^c, \end{aligned}$$

where $m = Ord(\langle c_i, d \rangle)$, and with each d the variables

$$in2_m \in IVar_{d^*}^c, \quad in3_m \in IVar_{Int^*}^c,$$

where $m = n^2 + 2n + Ord(d)$, and, finally, with c_i the variables

$$act1_m \in IVar_{c_i}^c, \quad act2_m \in IVar_{Int^*}^c,$$

where $m = Ord(c_i)$.

Finally, for each c we introduce a fresh variable $count \in IVar_{Int^*}^c$.

Definition 8.5

For $1 \leq i \leq n$ the set of variables, as introduced above, belonging to $IVar^{c_i}$, we denote by H^{c_i} .

Let $m = Ord(\langle c_i, d \rangle)$. Then for an arbitrary object of class c the variable $in1_m$ will record the order in which objects of class c_i communicated an object of class d to it. These communicated objects are stored in the order of their arrival in the variable $in2_m$. The variable $in3_m$ will record for each of the above communications the *local* time it occurs.

Analogously we have that for an arbitrary object of class c the variable $out1_m$ will record the order in which objects of class c_i received an object of class d from it. The communicated objects are stored again in the order of their arrival in the variable $out2_m$. The variable $out3_m$ will record the local time of each of these communications.

Let for $d \in C^+$ $m = n^2 + 2n + Ord(d)$. The variable $in2_m$ will record for each object of class c the sequence of objects of class d communicated by an unknown object. The variable $in3_m$ again will record the local time of each of these communications.

Let $m = Ord(c_i)$. Then for an arbitrary object of class c the variable $act1_m$ will record the order of objects of class c_i which have been created by it. The variable $act2_m$ will record for each of these activations its local time.

Finally, for each object of class c the integer variable $count$ will record its local time, i.e., this variable simply counts the number of communications and activations of this object.

As we introduce instance variables ranging over sequences only to be able to code histories we assume that $IVar_d^c = H_d^c$, for $d \in C^+$.

According to the above informal explanation we transform a program ρ , assuming $IVar(\rho) \cap \bigcup_c H^c = \emptyset$, to ρ' as follows:

Definition 8.6

First we transform an arbitrary I/O statement and new statement:

$$\begin{aligned}
 x_{c_i}^c ? y_d^c &\Rightarrow x_{c_i}^c ? y_d^c; count := count + 1; \\
 &\quad in1_m, in2_m, in3_m := in1_m \circ \langle x_{c_i}^c \rangle, in2_m \circ \langle y_d^c \rangle, in3_m \circ \langle count \rangle, \\
 &\quad \text{where } m = Ord(\langle c_i, d \rangle) \\
 x_{c_i}^c ! e_d^c &\Rightarrow x_{c_i}^c ! e_d^c; count := count + 1; \\
 &\quad out1_m, out2_m, out3_m := out1_m \circ \langle x_{c_i}^c \rangle, out2_m \circ \langle e_d^c \rangle, out3_m \circ \langle count \rangle \\
 &\quad \text{where } m = Ord(\langle c_i, d \rangle) \\
 ? y_d^c &\Rightarrow ? y_d^c; count := count + 1; \\
 &\quad in2_m, in3_m := in2_m \circ \langle y_d^c \rangle, in3_m \circ \langle count \rangle, \\
 &\quad \text{where } m = n^2 + 2n + Ord(d) \\
 x_d^c := new &\Rightarrow x_d^c := new; count := count + 1; \\
 &\quad act1_m, act2_m := act1_m \circ \langle x_d^c \rangle, act2_m \circ \langle count \rangle, \\
 &\quad \text{where } m = Ord(d)
 \end{aligned}$$

Here $x_1, \dots, x_k := e_1, \dots, e_k$ stands for $x_1 := e_1; \dots; x_k := e_k$. Let for an arbitrary statement S^c the result of applying the above transformation to S , defined by induction on the structure of S , be denoted by S' . Given a program $\rho = \langle U : S_n^{c_n} \rangle$, with $U = c_1 \leftarrow S_1^{c_1}, \dots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}}$, we define $\rho' = \langle U' : S_n'^{c_n} \rangle$, with $U' = c_1 \leftarrow S_1'^{c_1}, \dots, c_{n-1} \leftarrow S_{n-1}'^{c_{n-1}}$.

We introduce the following notation to refer to the history of an object of class c encoded by the values of the variables of H^c .

Definition 8.7

Let σ and α such that $\alpha \in \sigma^{(c)}$. We define $h_{\sigma, \alpha}$ to be the history encoded by the values of the variables of H^c of α , if such a history exists, otherwise $h_{\sigma, \alpha}$ is undefined.

8.2 A most general proof outline

Let $\rho = \langle U : S_n^{c_n} \rangle$, with $U = c_1 \leftarrow S_1^{c_1}, \dots, c_{n-1} \leftarrow S_{n-1}^{c_{n-1}}$, be a program such that $IVar(\rho) \cap \bigcup_c H^c = \emptyset$, and for which the formula $\{p \downarrow z_{c_n}\} \rho \{Q\}$ is valid, where

$IVar(p, Q) \cap \bigcup_c H^c = \emptyset$. We define a most general proof outline for ρ' , that is, we will define a global invariant I and for every occurrence of an arbitrary normal substatement S of ρ' local assertions $Pre(S)$ and $Post(S)$ such that for

$$A_k = \{\{Pre(S)\}S\{Post(S)\} : S \text{ an occurrence of a bracketed section of } S'^{c_k}\}$$

we have $A_k \vdash \{Pre(S'^{c_k})\}S'^{c_k}\{Post(S'^{c_k})\}$ and $Coop(A_1, \dots, A_n, I, z_{c_n})$. By an application of the rules (PR), (PC), (S1), (S2), (INIT) and (Aux) the derivability of $\{p \downarrow z_{c_n}\}\rho\{Q\}$ then can be shown to follow. First we modify the precondition p^{c_n} as follows:

Definition 8.8

We define

$$p'^{c_n} = p^{c_n} \wedge \bigwedge_{x \in W} (x \doteq z_x) \wedge Ihist,$$

where $W = IVar_{int}^{c_n} \cup IVar_{bool}^{c_n}$, z_x being a new logical variable uniquely associated with the instance variable x . These newly introduced variables are used to “freeze” that part of the initial state as specified by the integer and boolean variables. Furthermore *Ihist* stands for the assertion

$$count \doteq 0 \wedge \bigwedge_d \bigwedge_{x \in H_d^{c_n}} |x| \doteq 0.$$

The assertion *Ihist* initializes the history of the root-object.

We start with the definition of the global invariant. This global invariant describes all states σ for which there exists an intermediate configuration (X', σ') of a computation of ρ' such that σ and σ' agree with respect to the existing objects and with respect to the histories of these objects. Furthermore, in the configuration (X', σ') every existing object is executing *outside* a bracketed section, so no object is involved in a communication or the creation of some object.

Lemma 8.9

There exists a global assertion I , $IVar(I) \subseteq \bigcup_c H^c$, such that:

$\sigma, \omega \models I$ iff there exist configurations $(X_0, \sigma_0), (X_1, \sigma_1)$ such that:

- For some history $h : (X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$, where $Init_{\rho'}((X_0, \sigma_0))$, and for $\alpha \in \sigma_0^{(c_n)}$ we have $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'^{c_n}$.
- For an arbitrary c and $\alpha \in \sigma_1^{(c)}$ we have that $X_1(\alpha)$ is normal.
- For an arbitrary c we have $\sigma^{(c)} = \sigma_1^{(c)}$, and for every $\alpha \in \sigma^{(c)}$ we have $h_{\sigma, \alpha} = h_{\sigma_1, \alpha}$.

Proof

See section 9. □

The next lemma states the existence of a local assertion $Pre(R)$, R a normal substatement of S^{ci} . This local assertion describes all the local states $\theta = \langle \alpha, s \rangle$ for which there exists an intermediate configuration (X, σ) of a computation of ρ' such that α exists in σ and s equals $\sigma(\alpha)$, furthermore, in the configuration (X, σ) the object α is about to execute R .

Lemma 8.10

Let R be a normal substatement of S_i^{ci} . There exists a local assertion $Pre(R)$ such that:

$\theta^{ci}, \omega \models Pre(S)$ iff there exist configurations $(X_0, \sigma_0), (X_1, \sigma_1)$ such that:

- For some history h we have $(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$, where $Init_{\rho'}((X_0, \sigma_0))$, and for $\alpha \in \sigma_0^{(c_n)}$ we have $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'^{c_n}$.
- For some $\alpha \in \sigma_1^{(c_i)}$ we have $\langle \alpha, \sigma_1(\alpha) \rangle = \theta^{ci}$ and $X_1(\alpha) = Before(R, S_i^{ci})$.

Proof

See section 9. □

Analogously we have a lemma asserting the existence of a local assertion $Post(R)$, R a normal subprogram of, say, S_i^{ci} : Just substitute $After(R, S_i^{ci})$ for the phrase $Before(R, S_i^{ci})$.

Now we define our sets of assumptions.

Definition 8.11

Let

$$A_k = \{ \{ Pre(R) \} R \{ Post(R) \} : R \text{ a bracketed section occurring in } S_k^{c_k} \}.$$

We have the following lemma stating that from this set of assumptions A_k we can derive the local correctness formula $\{ Pre(S_k^{c_k}) \} S_k^{c_k} \{ Post(S_k^{c_k}) \}$.

Lemma 8.12

For the set of local correctness formulas A_k as defined above we have

$$A_k \vdash \{Pre(S_k^{lc})\} S_k^{lc} \{Post(S_k^{lc})\}.$$

Proof

Follows in a straightforward manner from the semantics of the assertions $Pre(R)$ and $Post(R)$, R a normal statement of S' , as described above, using lemma 7.9. \square

To show that these assumptions cooperate we need the following definition and lemma:

Definition 8.13

Let, for an arbitrary c , $B^c \subseteq \sigma^{(c)}$, and, for $\alpha \in B^c$, R_α be a bracketed section of S'^c or be the empty statement E . We call the set $\bigcup_c \{< \alpha, R_\alpha > : \alpha \in B^c\}$ (σ, ω) -*reachable* iff there exist configurations $(X_0, \sigma_0), (X_1, \sigma_1)$ such that:

- For some history h : $(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$, where $INIT_{p'}((X_0, \sigma_0))$, and for $\alpha \in \sigma_0^{(c_n)}$ we have $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'^{c_n}$.
- For an arbitrary c we have $B^c \subseteq \sigma_1^{(c)}$, for $\alpha \in B^c$ we have $Before(R_\alpha, S'^c) = X_1(\alpha)$ if $R_\alpha \neq E$, $X_1(\alpha) = E$, otherwise, and $\sigma(\alpha) = \sigma_1(\alpha)$.

If, additionally, we have for an arbitrary c that $\sigma^{(c)} = \sigma_1^{(c)}$ and for all $\alpha \in \sigma^{(c)}$ we have that $X_1(\alpha)$ is normal and $h_{\sigma, \alpha} = h_{\sigma_1, \alpha}$ we speak of (I, σ, ω) -*reachability*.

Note that $\sigma, \omega \models Pre(R) \downarrow z$, assuming R to be a bracketed section of S'^c and $z \in LogVar_c$, implies the (σ, ω) -*reachability* of $< \alpha, R >$, where $\alpha = \omega(z)$. We have the following lemma, which is called “the merging lemma”, about merging different computations into one computation.

Lemma 8.14

Let, for an arbitrary c , $B^c \subseteq \sigma^{(c)}$ be such that for $\alpha \in B^c$ and R_α (a bracketed section of S'^c or the empty statement E) we have that $< \alpha, R_\alpha >$ is (σ, ω) -*reachable* and $\sigma, \omega \models I$. It then follows that $\bigcup_c \{< \alpha, R_\alpha > : \alpha \in B^c\}$ is (I, σ, ω) -*reachable*.

Proof

By $\sigma, \omega \models I$ we have for some configurations $(X_0, \sigma_0), (X_1, \sigma_1)$:

- For some history h : $(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$, where $Init_{p'}((X_0, \sigma_0))$, and for $\alpha \in \sigma_0^{(c_n)}$ we have $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'^{c_n}$.

- For an arbitrary c and $\alpha \in \sigma_1^{(c)}$ we have that $X_1(\alpha)$ is normal.
- For an arbitrary c we have $\sigma^{(c)} = \sigma_1^{(c)}$, and for every $\alpha \in \sigma^{(c)}$ we have $h_{\sigma, \alpha} = h_{\sigma_1, \alpha}$.

By the (σ, ω) -reachability of $\langle \alpha, R_\alpha \rangle$ we have for some configurations $(X_\alpha, \sigma_\alpha), (X'_\alpha, \sigma'_\alpha)$:

- For some history $h_\alpha: (X_\alpha, \sigma_\alpha) \rightarrow^{h_\alpha} (X'_\alpha, \sigma'_\alpha)$, where $Init_{\rho'}((X_\alpha, \sigma_\alpha) \rangle)$, and for $\beta \in \sigma_\alpha^{(c_n)}$ we have $\langle \beta, \sigma_\alpha(\beta) \rangle, \omega \models p^{c_n}$.
- For $\alpha \in \sigma_\alpha^{(c)}$ we have $\sigma(\alpha) = \sigma'_\alpha(\alpha)$ and $X'_\alpha(\alpha) = Before(R_\alpha, S^{c'})$.

As a local computation of an object which is executing outside a bracketed section does not affect the history variables, i.e., the variables used to encode the local history, we may assume without loss of generality that there exists no configuration (X, σ) such that $(X_1, \sigma_1) \rightarrow^\epsilon (X, \sigma)$, so every existing object in (X_1, σ_1) is about to enter a bracketed section or has terminated. (Just execute the local process of an object until a bracketed section has reached or it has terminated.) It suffices to prove that for $\alpha \in B^c$ we have $(X_1(\alpha), \sigma_1(\alpha)) = (X'_\alpha(\alpha), \sigma'_\alpha(\alpha))$.

Now let $\alpha \in B^c$, with $c \neq c_n$: From the definition of the global transition system and the construction of ρ' it follows that $(S^{c'}, \nabla) \rightarrow^{h_{\sigma'_\alpha, \alpha}} (X'_\alpha(\alpha), \sigma'_\alpha(\alpha))$ and $(S^{c'}, \nabla) \rightarrow^{h_{\sigma, \alpha}} (X_1(\alpha), \sigma_1(\alpha))$. Now by $h_{\sigma'_\alpha, \alpha} = h_{\sigma, \alpha} = h_{\sigma_1, \alpha}$ (the first identity follows from $\sigma(\alpha) = \sigma'_\alpha(\alpha)$, the second from $\sigma, \omega \models I$), and the fact that the environment completely determines the local behaviour of an object we have $(X_1(\alpha), \sigma_1(\alpha)) = (X'_\alpha(\alpha), \sigma'_\alpha(\alpha))$.

Let $\alpha \in B^{c_n}$: From $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'$, $\langle \alpha, \sigma_\alpha(\alpha) \rangle, \omega \models p'$, and $Init_{\rho'}((X_0, \sigma_0), (X_\alpha, \sigma_\alpha))$ it follows that $\sigma_0(\alpha) = \sigma_\alpha(\alpha)$ (the assertion $\bigwedge_{x \in W} (x \doteq z_x)$ freezes the part of the initial state specified by the variables ranging over the standard objects, the predicate $Init_{\rho'}$ fixes the part of the initial state specified by the variables ranging over the non-standard objects, and, finally, the assertion $Ihist$ fixes the variables introduced to code the local history). So we can apply an argument similar to the one given above. \square

Given the merging lemma we are now ready to prove that the assumptions introduced above cooperate. First we show how to discharge assumptions about bracketed sections containing a new-statement.

Lemma 8.15

Let $R = x := \text{new}; R_1$ be a bracketed section occurring in $S_i^{c_i}$, assuming the type of the variable x to be c_j , then:

$$\vdash \{I \wedge \text{Pre}(R) \downarrow z\}(z, R) \{I \wedge \text{Post}(R) \downarrow z \wedge \text{Pre}(S_j^{c_j}) \downarrow z'\}[z.x/z']\}.$$

where $z \in LVar_{c_i}$ and $z' \in LVar_{c_j}$ are two distinct new variables.

Proof

Let $[act]$ abbreviate the substitution corresponding to the updating of the local history as described by R_1 :

$$[z.act1_j \circ < z.x >, z.act2_j \circ < z.count + 1 >, z.count + 1/z.act1_j, z.act2_j, z.count].$$

By the axioms (IASS) and (NEW) it suffices to show that:

$$\begin{aligned} & \models I \wedge \text{Pre}(R) \downarrow z \rightarrow \\ & (I \wedge \text{Post}(R) \downarrow z \wedge \text{Pre}(S_j^{c_j}) \downarrow z'[z.x/z'])[act][z''/z.x][\text{new}/z''] \end{aligned}$$

where $z'' \in LVar_{c_j}$ is a new variable: Let $\sigma, \omega \models I \wedge \text{Pre}(R) \downarrow z$. Now $\sigma, \omega \models \text{Pre}(R) \downarrow z$ implies the (σ, ω) - *reachability* of $\langle \omega(z), R \rangle$. An application of the merging lemma gives us the (I, σ, ω) - *reachability* of $\langle \omega(z), R \rangle$. So there exist configurations $(X_1, \sigma_1), (X_2, \sigma_2)$ such that

- For some history h : $(X_1, \sigma_1) \rightarrow^h (X_2, \sigma_2)$, where $\text{Init}_{p'}((X_1, \sigma_1))$, and for $\alpha \in \sigma_1^{(c_n)}$ we have $\langle \alpha, \sigma(\alpha) \rangle, \omega \models p'^{c_n}$.
- For an arbitrary c we have $\sigma^{(c)} = \sigma_2^{(c)}$, and for $\alpha \in \sigma^{(c)}$ we have that $X_1(\alpha)$ is normal and $h_{\sigma, \alpha} = h_{\sigma_2, \alpha}$.
- For $\alpha = \omega(z)$ we have $\sigma(\alpha) = \sigma_2(\alpha)$ and $\text{Before}(R, S_i^{c_i}) = X_2(\alpha)$.

Let $\sigma' \in \mathcal{I}[(z, R)](\omega)(\sigma)$. It then follows that for σ_3 such that

$$\begin{aligned} \sigma_3^{(c)} &= \sigma_2^{(c)} \cup \{\beta\} \quad c = c_j \\ &= \sigma_2^{(c)} \quad \text{otherwise,} \end{aligned}$$

with $\beta = \sigma'(\alpha)(x)$, and $\sigma_{3(2)} = \sigma_{2(2)}\{\sigma'(\alpha)/\alpha\}\{\nabla/\beta\}$, with $\alpha = \omega(z)$, we have $(X_1, \sigma_1) \rightarrow^{h'} (X_3, \sigma_3)$, where $h' = h \circ \langle \alpha, \beta \rangle$ and $X_3 = X_2\{\text{After}(R, S_i^{c_i})/\alpha\}$.

By $(X_1, \sigma_1) \rightarrow^{h'} (X_3, \sigma_3)$ we have $\sigma', \omega \models I$. Furthermore it follows that $\sigma', \omega \models \text{Post}(R) \downarrow z$ and $\sigma', \omega\{\beta/z'\} \models \text{Pre}(S_j^{c_j}) \downarrow z'$, or, equivalently, $\sigma', \omega \models \text{Pre}(S_j^{c_j}) \downarrow z'[z.x/z']$.

Summerizing we have

$$\sigma', \omega \models (I \wedge \text{Post}(R) \downarrow z \wedge \text{Pre}(S'^{c_j}) \downarrow z'[z.x/z']).$$

From which in turn it is not difficult to derive by an application of the lemmas 7.1 and 7.3 that:

$$\sigma, \omega \models (I \wedge \text{Post}(R) \downarrow z \wedge \text{Pre}(S'^{c_j}) \downarrow z'[z.x/z'])[act][z''/z.x][new/z'']. \quad \square$$

Next we show how to discharge assumptions about matching bracketed sections.

Lemma 8.16

For two arbitrary matching bracketed sections R_1 and R_2 , occurring in, say, S'^{c_i}, S'^{c_j} , $1 \leq i < n$, $1 \leq j \leq n$, we have:

$$\begin{array}{c} \{I \wedge \text{Pre}(R_1) \downarrow z \wedge \text{Pre}(R_2) \downarrow z'\} \\ \vdash \quad (z, R_1) \parallel (z', R_2) \\ \{I \wedge \text{Post}(R_1) \downarrow z \wedge \text{Post}(R_2) \downarrow z'\} \end{array}$$

where $z \in LVar_{c_i}$ and $z' \in LVar_{c_j}$ are two new distinct variables.

Proof

We prove the following case, the other ones are treated in a similar way: Let $R_1 = x!e; R'_1$, $R_2 = ?y; R'_2$, and $i \neq j$. Let $[in]$ abbreviate the substitution corresponding to the update of the local history as given by R'_1 :

$$\left[\begin{array}{l} z'.in1_{k0} < z'.y >, z'.in2_{k0} < z'.count + 1 >, z'.count + 1 / \\ z'.in1_k, z'.in2_k, z'.count. \end{array} \right].$$

And $[out]$ abbreviate the substitution corresponding to the update of the local history as given by R'_2 :

$$\left[\begin{array}{l} z.out1_{l0} < z.x >, z.out2_{l0} < e \downarrow z >, z.out3_{l0} < z.count + 1 >, z.count + 1 / \\ z.out1_l, z.out2_l, z.out3_l, z.count. \end{array} \right].$$

By the axioms (COMM) and (IASS), the rules (SR2), (PAR1) and (PAR2) it suffices to show that:

$$\begin{array}{l} \models I \wedge \text{Pre}(R_1) \downarrow z \wedge \text{Pre}(R_2) \downarrow z' \wedge z.x = z' \wedge z.x \neq \text{nil} \rightarrow \\ (I \wedge \text{Post}(R_1) \downarrow z \wedge \text{Post}(R_2) \downarrow z')[in][out][e \downarrow z/z'.y]. \end{array}$$

Let $\sigma, \omega \models (I \wedge \text{Pre}(R_1) \downarrow z \wedge \text{Pre}(R_2) \downarrow z' \wedge z.x = z' \wedge z.x \neq \text{nil})$. It follows that $\langle \alpha, R_1 \rangle$ and $\langle \beta, R_2 \rangle$ are (σ, ω) -reachable, where $\alpha = \omega(z)$ and $\beta = \omega(z')$. From lemma 8.14 then we infer the (I, σ, ω) -reachability of $\{\langle \alpha, R_1 \rangle, \langle \beta, R_2 \rangle\}$. So there exist configurations $(X_1, \sigma_1), (X_2, \sigma_2)$ such that

- For some history h : $(X_1, \sigma_1) \rightarrow^h (X_2, \sigma_2)$, $Init_{\rho'}((X_1, \sigma_1))$, and for $\alpha \in \sigma_1^{(c_n)}$ we have $\langle \alpha, \sigma(\alpha) \rangle, \omega \models p'^{c_n}$.
- For an arbitrary c we have $\sigma^{(c)} = \sigma_2^{(c)}$ and for $\alpha \in \sigma^{(c)}$ we have that $X_2(\alpha)$ is normal and $h_{\sigma, \alpha} = h_{\sigma_2, \alpha}$.
- We have $Before(R_1, S_i^{c_i}) = X_2(\alpha)$, $Before(R_2, S_j^{c_j}) = X_2(\beta)$, and $\sigma(\alpha) = \sigma_2(\alpha)$, $\sigma(\beta) = \sigma_2(\beta)$.

Let $\sigma' \in \mathcal{I}[(z, R_1) \parallel (z', R_2)](\omega)(\sigma)$. Note that such a σ' exists because we have $\sigma, \omega \models z.x \doteq z' \wedge z.x \neq \text{nil}$.

By the definition of the global transition system it follows that for σ_3 , with $\sigma_3^{(c)} = \sigma_2^{(c)}$, c arbitrary, and $\sigma_{3(2)} = \sigma_{2(2)}\{\sigma'(\alpha)/\alpha\}\{\sigma'(\beta)/\beta\}$ we have $(X_1, \sigma_1) \rightarrow^{h'} (X_3, \sigma_3)$, where $h' = h \circ \langle \alpha, \beta, \gamma \rangle$, for some γ , and $X_3 = X_2\{After(R_1, S_i^{c_i})/\alpha\}\{After(R_2, S_j^{c_j})/\beta\}$.

From $(X_1, \sigma_1) \rightarrow^{h'} (X_3, \sigma_3)$ it then follows that $\sigma', \omega \models I$. Furthermore we have $\sigma', \omega \models Post(R_1) \downarrow z$ and $\sigma', \omega \models Post(R_2) \downarrow z'$. Summerizing we have:

$$\sigma', \omega \models (I \wedge Post(R_1) \downarrow z \wedge Post(R_2) \downarrow z').$$

From which in turn we derive by applying lemma 7.1 that

$$\sigma, \omega \models (I \wedge Post(R_1) \downarrow z \wedge Post(R_2) \downarrow z')[in][out][e \downarrow z/z'.y].$$

□

We summerize the above by the following theorem.

Theorem 8.17

The following formula about ρ' , which is called the most general correctness formula about ρ' , is derivable:

$$\begin{aligned} & \{Pre(S_n^{c_n}) \downarrow z_{c_n}\} \\ & \quad \rho' \\ & \{I \wedge \bigwedge_{i \neq n} \forall z_{c_i} Post(S_i^{c_i}) \downarrow z_{c_i} \wedge Post(S_n^{c_n}) \downarrow z_{c_n}\}. \end{aligned}$$

Proof

By lemma 8.12 we have

$$A_k \vdash \{Pre(S_k^{c_k})\} S_k^{c_k} \{Post(S_k^{c_k})\}.$$

Furthermore it is not difficult to prove that

$$\models \text{Pre}(S_n^{c_n}) \downarrow z_{c_n} \wedge \forall z'_{c_n} (z'_{c_n} = z_{c_n}) \wedge \bigwedge_{1 \leq i < n} (\forall z \text{ false}) \rightarrow I.$$

From this and the lemmas 8.15 and 8.16 it follows that $\text{Coop}(A_1, \dots, A_n, I, z_{c_n})$. Applying the rule (PR) then finishes the proof. \square

We are now ready for the completeness theorem.

Theorem 8.18

The valid correctness formula $\{p \downarrow z_{c_n}\} \rho \{Q\}$ is derivable (assuming $\text{IVar}(p, Q, \rho) \cap \bigcup_c H^c = \emptyset$):

$$\vdash \{p \downarrow z_{c_n}\} \rho \{Q\}.$$

Proof

By the previous theorem we have the derivability of the correctness formula

$$\frac{\{ \text{Pre}(S_n^{c_n}) \downarrow z_{c_n} \}}{\rho'} \{ I \wedge \bigwedge_{i \neq n} \forall z_{c_i} \text{Post}(S_i^{c_i}) \downarrow z_{c_i} \wedge \text{Post}(S_n^{c_n}) \downarrow z_{c_n} \}.$$

It is not difficult to prove that $\models p^{c_n} \wedge \bigwedge_{x \in W} x \doteq \text{nil} \rightarrow \text{Pre}(S_n^{c_n})$, where $W = \bigcup_c \text{IVar}_c^{c_n}$. Next we prove

$$\models I \wedge \bigwedge_{i \neq n} \forall z_{c_i} \text{Post}(S_i^{c_i}) \downarrow z_{c_i} \wedge \text{Post}(S_n^{c_n}) \downarrow z_{c_n} \rightarrow Q.$$

Let the antecedent of the implication be true with respect to some logical environment ω and some global state σ . Note that for an arbitrary $c, \alpha \in \sigma^{(c)}$, we then have that $\langle \alpha, E \rangle$ is (σ, ω) -reachable. Applying lemma 8.14 thus gives us the (I, σ, ω) -reachability of $\bigcup_c \{ \langle \alpha, E \rangle : \alpha \in \sigma^{(c)} \}$. It thus follows that $(X_1, \sigma_1) \rightarrow^h (X_2, \sigma)$ for some history h , where $\text{Init}_{\rho'}((X_1, \sigma_1))$, $\text{Final}_{\rho'}((X_2, \sigma))$, and $\sigma_1, \omega \models p \downarrow z_{c_n}$. Now the validity of the formula $\{p \downarrow z_{c_n}\} \rho \{Q\}$ implies that of $\{p \downarrow z_{c_n}\} \rho' \{Q\}$ (note that $\text{IVar}(p, Q, \rho) \cap \bigcup_c H^c = \emptyset$, and that the updating of the history variables does not affect the flow of control of ρ), from which we conclude $\sigma, \omega \models Q$.

By the consequence rule and the rule (INIT) we thus have

$$\vdash \{p' \downarrow z_{c_n}\} \rho' \{Q\}.$$

Applying the substitution rules (S1) and (S2), substituting every logical variable z_x by the corresponding instance variable x , substituting the instance variable *count* by

0, and every variable $x \in H_{d^*}^{c_n}$, $d \in C^+$, by nil, the empty sequence, then gives us, after an trivial application of the consequence rule,

$$\vdash \{p \downarrow z_{c_n}\} \rho' \{Q\}.$$

An application of the rule (Aux) then finishes the proof. \square

9 Expressibility

In this section we show that the assertions I , $Pre(R)$ and $Post(R)$, as defined in the section on the completeness of the proof system, can indeed be expressed in our assertion languages. We assume throughout this section the sets C and $IVar$ to be finite. We start with a description of the *coding* techniques we will use to express these assertions.

9.1 Coding techniques

We will have to use some coding mechanism to *arithmetize* the semantics of programs and assertions. However we will not go into the details of the construction of the mechanism we need. We will just assume such a coding mechanism to exist, and list some of its properties we shall make use of. For the details of the coding mechanism we refer to [TZ]. Let ρ' be the program defined in the section on completeness (see definition 8.6), relative to which the assertions I , $Pre(R)$ and $Post(R)$ are defined.

For an arbitrary statement S , variable x , the code of S and x will be denoted by $[S], [x] \in N$, respectively.

For an arbitrary d we assume given an injection $\llbracket \cdot \rrbracket_d \in O_{\perp}^d \rightarrow N$ such that $\llbracket \perp \rrbracket_d = 0$. For $d = \text{Int}$ we assume $\llbracket \cdot \rrbracket_d$ to be surjective too.

Furthermore we assume given for an arbitrary d an injection $\llbracket \cdot \rrbracket_{d^*} \in O^{d^*} \rightarrow N$ such that for $\alpha = \langle \beta_1, \dots, \beta_n \rangle \in O^{d^*}$, with $d \in C$, we have $\llbracket \alpha \rrbracket_{d^*} = \langle \llbracket \beta_1 \rrbracket_d, \dots, \llbracket \beta_n \rrbracket_d \rangle$.

The code of an arbitrary global state σ and a global configuration (X, σ) will be denoted by $[\sigma], [(X, \sigma)] \in N$, respectively.

We assume the representability in the local assertion language of $\llbracket \cdot \rrbracket_{\text{Int}^*}$ and of the coding functions $\llbracket \cdot \rrbracket_{\text{Int}}$ and $\llbracket \cdot \rrbracket_{\text{Bool}}$. So, for example, we assume the existence of an assertion $p(z_1, z_2)$, where $z_1 \in LVar_{d^*}$, $z_2 \in LVar_d$, $d = \text{Int}$, such that for an arbitrary ω, θ we have

$$\theta, \omega \models p(z_1, z_2) \text{ iff } [\omega(z_1)]_{d^*} = \omega(z_2).$$

As we will explain in the following subsection the coding function $[]_c$ is *not* representable in the local assertion language or in the global assertion language. For example there exists no local assertion $p(x, z)$, the type of the variable x being c and that of the variable z being Int , such that for every θ and ω :

$$\theta, \omega \models p(x, z) \text{ iff } [\theta(x)]_c = \omega(z).$$

However we will see that we do not need these functions to be representable in the assertion language.

Next we present a list of relations between natural numbers which we assume to be recursive, and thus definable in the local assertion language:

Definition 9.1

We define

- $\text{Len}_d(n) = m$ iff there exists a sequence $\alpha \in \mathbf{O}^{d^*}$ such that $[\alpha]_{d^*} = n$ and the length of α equals m .
- $\text{Conc}_d(n, m) = k$ iff there exist $\alpha, \beta \in \mathbf{O}^{d^*}$ such that $n = [\alpha]_{d^*}$, $m = [\beta]_{d^*}$, and $k = [\alpha \circ \beta]_{d^*}$.
- $\text{Ell}_d(n, m) = k$ iff there exist $\alpha \in \mathbf{O}^{d^*}$, $\beta \in \mathbf{O}_{\perp}^{\text{Int}}$ such that $[\alpha]_{d^*} = n$, $[\beta]_{\text{Int}} = m$, and $[\alpha(\beta)]_d = k$.
- $\text{Act}_c(n, m)$ iff there exist a global state σ , $\alpha \in \sigma^{(c)}$ such that $m = [\sigma]$, $[\alpha]_c = n$.
- $\text{Val}_a^c(k, l, m) = n$ iff there exist a program variable x_a^c , a global state σ , $\alpha \in \sigma^{(c)}$ such that $[\sigma] = m$, $[\alpha]_c = k$, $[x_a^c] = l$, and $[\sigma(\alpha)(x_a^c)]_a = n$.
- $\text{Trans}(n, m)$ iff there exist global configurations (X_0, σ_0) , (X, σ) such that $[(X_0, \sigma_0)] = n$, $[(X, \sigma)] = m$, $\text{Init}_{\rho'}((X_0, \sigma_0))$, and for some history h $(X_0, \sigma_0) \rightarrow^h (X, \sigma)$.
- $\text{Norm}_c(n, m)$ iff there exist a global configuration (X, σ) , $\alpha \in \sigma^{(c)}$ such that $[\alpha]_c = n$, $[(X, \sigma)] = m$, and $X(\alpha)$ is normal.
- $\text{Before}_c(k, l) = m$ iff there exist statements R^c and S^c such that $[R^c] = k$, $[S^c] = l$, and $[\text{Before}(R^c, S^c)] = m$.
- $\text{After}_c(k, l) = m$ iff there exist statements R^c and S^c such that $[R^c] = k$, $[S^c] = l$, and $[\text{After}(R^c, S^c)] = m$.
- $\text{Prog}_c(n, m) = k$ iff there exist a global configuration (X, σ) , $\alpha \in \sigma^{(c)}$ such that $[(X, \sigma)] = m$, $[\alpha]_c = n$, and $[X(\alpha)] = k$.
- $\text{State}(n) = m$ iff there exists a global configuration (X, σ) such that $[(X, \sigma)] = n$ and $[\sigma] = m$.

In the sequel we identify the relations defined above with their representations in the local assertion language.

Next we code the truth relation $\sigma, \omega \models p^c \downarrow z_c, p^c$ a local assertion. We do so by translating a local expression l^c and a local assertion p^c to an *arithmetical* one, which we denote by $l^c[u, v]$ and $p^c[u, v]$, where u, v are fresh logical integer variables. We call an expression (assertion) arithmetical if only logical variables ranging over (sequences of) integers occur in it. (So occurrences of instance variables are not allowed.) The idea is that the translated expression (assertion) “speaks” about a state in terms of the arithmetical structure of its code number. The variable u will be interpreted as the code of the current state, and the variable v as the code of the object with respect to which the expression (assertion) is evaluated. The translation runs as follows:

Definition 9.2

We define

- $z_d[u, v] = [z_d]_d$
- $x_a^c[u, v] = Val(v, [x_a^c], u)$
- $nil[u, v] = 0$
- $self[u, v] = v$
- $\underline{n}[u, v] = [\underline{n}]_{Int}$
- $true[u, v] = [true]_{Bool}$
- $false[u, v] = [false]_{Bool}$
- $(e_1^c : e_2^c)[u, v] = Elt(e_1^c[u, v], e_2^c[u, v])$
- $< e^c > [u, v] = [< e^c[u, v] >]_{Int^*}$
- $(e_1^c \circ e_2^c)[u, v] = Conc(e_1^c[u, v], e_2^c[u, v])$
- $|e^c|[u, v] = [Len(e^c[u, v])]_{Int}$
- $(e_1^c \doteq e_2^c)[u, v] = e_1^c[u, v] \doteq e_2^c[u, v]$
- $(\neg p^c)[u, v] = \neg(p^c[u, v])$
- $(p_1^c \wedge p_2^c)[u, v] = p_1^c[u, v] \wedge p_2^c[u, v]$
- $(\exists z_a p^c)[u, v] = \exists z_a(p^c[u, v])$

Note that by definition of the local assertion language we have in the first clause above that $d = Int, Bool$. With respect to the last clause we have $a = d, d^*, d = Int, Bool$. We have the following semantical property of this translation:

Lemma 9.3

For an arbitrary $\sigma, \alpha \in \sigma^{(c)}$, ω such that $OK(\sigma, \omega)$ we have

$$[\mathcal{L}][l_a^c \downarrow z_c](\omega\{\alpha/z_c\})(\sigma)_a = \mathcal{L}[l_a^c[u, v]](\omega\{[\sigma], [\alpha]_c/u, v\})(\sigma),$$

and

$$\mathcal{A}[p^c \downarrow z_c](\omega\{\alpha/z_c\})(\sigma) = \mathcal{A}[p^c[u, v]](\omega\{[\sigma], [\alpha]_c/u, v\})(\sigma).$$

This lemma states that taking the code of the object denoted by the expression l yields the same result as when we evaluate the translation of l into an arithmetical expression, interpreting the variables u and v as the current object and the current state, respectively. With respect to assertions this lemma expresses a corresponding proposition. Note that in fact the state with respect to which we evaluate the translated expression (assertion) is irrelevant because there do not occur instance variables in the translated expression (assertion).

Proof

Straightforward induction on the structure of l_a^c and p^c . □

9.2 Object-space isomorphisms

To proceed we next introduce the notion of an object-space isomorphism, an *osi* for short.

Definition 9.4

An *object-space isomorphism* (*osi*) is a family of functions $f = \langle f^d \rangle_{d \in G^+}$, where $f^d \in \mathbf{O}_\perp^d \rightarrow \mathbf{O}_\perp^d$ is a bijection, $f^d(\perp) = \perp$ and f^d , for $d = \text{Int}, \text{Bool}$, is the identity mapping.

Given an *osi* we define the isomorphic image of a local state as follows:

Definition 9.5

For an arbitrary local state θ^c , *osi* f we define the local state $f(\theta^c)$ as follows:

- $f(\theta^c)_{(1)} = f^c(\theta_{(1)}^c)$.
- For an arbitrary variable $x \in IVar_d^c$ we have $f(\theta^c)_{(2)}(x) = f^d(\theta_{(2)}^c(x))$.

Given an *osi* we define next the isomorphic image of a global state.

Definition 9.6

For an arbitrary global state σ , *osi* f we define the global state $f(\sigma)$ as follows:

- For an arbitrary c we have $f(\sigma)(c) = f^c(\sigma(c))$.
- For an arbitrary c , $\alpha \in \sigma(c)$ we have $f(\sigma)(\alpha) = f^d(\sigma(g^c(\alpha)))$, where $g = f^{-1}$ and f^{-1} denotes the *inverse* of f : $f^{-1} = (f^d)^{-1} >_d$.

Finally, given an *osi* we define the isomorphic image of a history.

Definition 9.7

Let h be a history and f be an *osi* we define $f(h)$ as follows:

$$\begin{aligned}
 f(h) &= \epsilon \\
 &\text{if } h = \epsilon \\
 &= f(h_1) \circ < f^{d_i}(\alpha), f^{d_j}(\beta) > \\
 &\text{if } h = h_1 \circ < \alpha, \beta >, \alpha \in O_{\perp}^{d_i}, \beta \in O_{\perp}^{d_j} \\
 &= f(h_1) \circ < f^{d_i}(\alpha), f^{d_j}(\beta), f^{d_k}(\gamma) > \\
 &\text{if } h = h_1 \circ < \alpha, \beta, \gamma >, \alpha \in O_{\perp}^{d_i}, \beta \in O_{\perp}^{d_j}, \gamma \in O_{\perp}^{d_k}
 \end{aligned}$$

The following theorem states that isomorphic states cannot be distinguished by the assertion languages.

Theorem 9.8

For an arbitrary *osi* f , local state θ^c , logical environment ω and local expression l_a^c , local assertion p^c we have

- $f^a(\mathcal{L}[l_a^c](\omega)(\theta^c)) = \mathcal{L}[l_a^c](f(\omega))(f(\theta^c))$.
- $\mathcal{A}[p^c](\omega)(\theta^c) = \mathcal{A}[p^c](f(\omega))(f(\theta^c))$.

where $f(\omega)(z) = f^a(\omega(z))$, $z \in LVar_a$. Furthermore for an arbitrary global state σ , logical environment ω such that $OK(\omega, \sigma)$, global expression g_a and global assertion P we have

- $f^a(\mathcal{G}[g_a](\omega)(\sigma)) = \mathcal{G}[g_a](f(\omega))(f(\sigma))$.
- $\mathcal{A}[P](\omega)(\sigma) = \mathcal{A}[P](f(\omega))(f(\sigma))$.

Proof

Induction on the structure of l_a^c, p^c, g_a and P respectively. \square

The following theorem states that our transition system is closed under isomorphic images.

Theorem 9.9

For two arbitrary global configurations $(X_0, \sigma_0), (X_1, \sigma_1)$, *osi* f and history h we have if

$$(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$$

then

$$(f(X_0), f(\sigma_0)) \rightarrow^{f(h)} (f(X_1), f(\sigma_1)),$$

where $f(X)(\alpha) = X(f^{-1}(\alpha))$.

Proof

It is not difficult to see that it suffices to prove the corresponding proposition for the local transition system: If

$$(S_1, \theta_1) \rightarrow^r (S_2, \theta_2)$$

then

$$(S_1, f(\theta_1)) \rightarrow^{f(r)} (S_2, f(\theta_2)).$$

This is proved by a straightforward case analysis, making use of theorem 9.8. \square

9.3 Coding the global invariant

Now we are ready to show how to express the global invariant as defined in the section on completeness. One of the main difficulties with expressing the global invariant is caused by the fact that we cannot express directly for example the relation $\sigma^{(c)} = \sigma'^{(c)}$, σ and σ' arbitrary, in terms of the code for σ' . More precisely, there exists no assertion $P(z)$ such that for every state σ and environment ω , with $OK(\sigma, \omega)$ and $\omega(z) = [\sigma']$, for some state σ' , we have

$$\sigma, \omega \models P(z) \text{ iff } \sigma^{(c)} = \sigma'^{(c)}.$$

This is an immediate consequence of theorem 9.8: We have

$$\sigma, \omega \models P(z) \text{ iff } f(\sigma), f(\omega) \models P(z),$$

for an arbitrary *osi* f . But it is not the case that for every *osi* f we have $f(\sigma)^{(c)} = \sigma'^{(c)}$ (note that $f(\omega)(z) = [\sigma']$). However using theorem 9.9 we will see that we do not need to express this coding relation directly.

To proceed we first introduce some new logical variables.

Definition 9.10

For an arbitrary c let $bij_c \in LVar_c^*$. Furthermore let \overline{bij} denote a sequence of these variables.

Given a state σ and an *osi* f these variables \overline{bij} will be used to code the restriction of f to the existing objects of σ in the following way:

Definition 9.11

For an arbitrary state σ , environment ω , with $OK(\omega, \sigma)$, and *osi* f we define $Code(\omega, \sigma, f)$ iff for an arbitrary c , $\alpha \in \sigma^{(c)}$ we have $\omega(bij_c)([f^c(\alpha)]_c) = \alpha$.

So every existing object of an arbitrary class c is stored in the sequence denoted by bij_c at a position which equals the code of its image under f .

Now we are ready to define an assertion $Bij(H^c, z_c, n, m)$ which expresses for two arbitrary states σ, σ_1 , $\alpha \in \sigma^{(c)}$, and *osi* f the relation $f(h_{\sigma, \alpha}) = h_{\sigma_1, f^c(\alpha)}$. The idea is that the logical variable z_c is interpreted as the object α , the integer variable n as the code of $f^c(\alpha)$, and the integer variable m as the code of σ_1 . In the formulation of the assertion $Bij(H^c, z_c, n, m)$ we will assume that the variables of \overline{bij} code the restriction of f to the existing objects of σ .

Definition 9.12

Let n and m be two logical integer variables. We define

$$Bij(H^c, z_c, n, m) = \bigwedge_{x \in H^c} P(z_c.x, n, m),$$

where

$$P(z_c.x, n, m) =$$

$$\begin{aligned} \forall 1 \leq j \leq |z_c.x| \quad (& z_c.x : j \doteq \text{nil} \rightarrow \text{Elt}(\text{Val}(n, [x], m), [j]_{\text{Int}}) \doteq 0 \wedge \\ & \forall k(z_c.x : j \doteq bij_{c_i} : k \neq \text{nil} \rightarrow \text{Elt}(\text{Val}(n, [x], m), [j]_{\text{Int}}) \doteq k \wedge \\ & |x| \doteq \text{Len}(\text{Val}(n, [x], m)) \\ &) \end{aligned}$$

Here we assume the type of x to be c_i^* . If on the other hand the type of x is Int^* we have

$$P(z_c.x, n, m) =$$

$$\begin{aligned} \forall 1 \leq j \leq |z_c.x| \quad (& z_c.x : j \neq \text{nil} \rightarrow \text{Elt}(\text{Val}(n, [x], m), [j]_{\text{Int}}) \doteq [z_c.x : j]_{\text{Int}} \wedge \\ & z_c.x : j \doteq \text{nil} \rightarrow \text{Elt}(\text{Val}(n, [x], m), [j]_{\text{Int}}) \doteq 0 \wedge \\ & |z_c.x| \doteq \text{Len}(\text{Val}(n, [x], m)) \\ &) \end{aligned}$$

The meaning of the assertion $P(z_c.x, n, m)$, with c_i^* the type of x , can be explained as follows: Suppose that the variable n is interpreted as the code of the object $\alpha = \omega(z_c)$ and the variable m as the code of the state σ such that $\alpha \in \sigma^{(c)}$. The assertion $P(z_c.x, n, m)$ then states that every element of the sequence denoted by $\sigma(\alpha)(x)$ occurs at a position in the sequence denoted by bij_{c_i} which equals its code. Formally we have the following lemma about this semantical property of the assertion defined above.

Lemma 9.13

Let $Code(\omega, \sigma, f)$, $\alpha \in \sigma^{(c)}$ and σ' such that $f(\sigma) \approx \sigma'$. (Here $\sigma_1 \approx \sigma_2$ holds iff for every c we have $\sigma_1^{(c)} = \sigma_2^{(c)}$.) We have

$$\sigma, \omega\{\alpha/z_c, [f^c(\alpha)]_c/n, [\sigma']/m\} \models Bij(H^c, z_c, n, m) \text{ iff } f(h_{\sigma, \alpha}) = h_{\sigma', f(\alpha)}.$$

Proof

It is not difficult to prove that

$$\begin{aligned} f(h_{\sigma, \alpha}) &= h_{\sigma', f(\alpha)} \text{ iff} \\ \text{for every } d, x \in H_{d^*}^c : f(\sigma(\alpha)(x)) &= \sigma'(f^c(\alpha))(x). \end{aligned}$$

Furthermore from $Code(\omega, \sigma, f)$ and $f(\sigma) \approx \sigma'$ it follows by a straightforward but slightly tedious argument that

$$f(\sigma(\alpha)(x)) = \sigma'(f^c(\alpha))(x) \text{ iff } \sigma, \omega\{\alpha/z_c, [f^c(\alpha)]_c/n, [\sigma']/m\} \models P(z_c.x, n, m).$$

□

Now we can express the global invariant as follows.

Lemma 9.14

We define

$$\begin{aligned} I = \\ \exists n, m, k \quad (& Trans(n, m) \wedge p'[u, v][State(n), k/u, v] \wedge Act_{c_n}(k, State(n)) \wedge \\ & \bigwedge_c \forall i (Act_c(i, m) \rightarrow Norm_c(i, m)) \wedge \\ & \exists \overline{bij} \quad (\bigwedge_c \forall z_c \exists ! 1 \leq j \leq |bij_c| (bij_c : j = z_c) \wedge \\ & \quad \bigwedge_c \forall i (Act(i, State(m)) \leftrightarrow bij_c : i \neq nil) \wedge \\ & \quad \bigwedge_c \forall 1 \leq j \leq |bij_c| Bij(H^c, bij_c : j, j, State(m)) \\ &) \\ &) \end{aligned}$$

(Here “ $\exists!$ ” is interpreted as “there exists an unique”, which can be expressed by the usual quantifiers.) Note that the quantification $\exists n, m$ corresponds to the phrase “there exist configurations $(X_0, \sigma_0), (X_1, \sigma_1)$ ”. The assertion $Trans(n, m)$ then corresponds to “there exists an history h such that $(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$ and $Init_{p'}((X_0, \sigma_0))$ ”. The assertion $\exists k(p'[u, v][State(n), k/u, v] \wedge Act_{c_n}(k, State(n)))$ translates the phrase “for $\alpha \in \sigma_0^{(c_n)}$ we have $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'$ ”. The assertion $\forall i(Act_c(i, m) \rightarrow Norm_c(i, m))$ corresponds to “for an arbitrary c and $\alpha \in \sigma_1^{(c)}$ we have that $X_1(\alpha)$ is normal”. Finally, the assertion

$$\begin{aligned} \exists \overline{bij} \quad (& \bigwedge_c \forall z_c \exists! 1 \leq j \leq |bij_c| bij_c : j = z_c \wedge \\ & \bigwedge_c \forall i Act(i, State(m)) \leftrightarrow bij_c : i \neq nil \wedge \\ & \bigwedge_c \forall 1 \leq j \leq |bij_c| Bij(H^c, bij_c : j, j, State(m)) \\ &) \end{aligned}$$

will correspond to the phrase “for an arbitrary c we have $\sigma^{(c)} = \sigma_1^{(c)}$ and for every $\alpha \in \sigma^{(c)}$ we have $h_{\sigma, \alpha} = h_{\sigma_1, \alpha}$ ”.

Proof

Let $\sigma, \omega \models I$. So there exist $\gamma_1, \gamma_2, \gamma_3 \in \mathbb{N}$ such that $\sigma, \omega' \models I$, where $\omega' = \omega\{\gamma_1, \gamma_2, \gamma_3/n, m, k\}$.

By $\sigma, \omega' \models Trans(n, m)$ we have $Init_{p'}((X_0, \sigma_0))$ and $(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$, for some history h , where $[(X_0, \sigma_0)] = \gamma_1$ and $[(X_1, \sigma_1)] = \gamma_2$.

From $\sigma, \omega' \models p'[u, v][State(n), k/u, v] \wedge Act_{c_n}(k, State(n))$ by an application of lemma 9.3 it follows that $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'$, where $\alpha \in \sigma_0^{(c_n)}$.

By $\sigma, \omega' \models \bigwedge_c \forall i(Act_c(i, m) \rightarrow Norm_c(i, m))$ we have that $X_1(\alpha)$ is normal for every $\alpha \in \sigma_1^{(c)}$, c arbitrary.

Now let $\alpha_i \in \mathbf{O}^{c_i}$ such that for $\omega'' = \omega\{\alpha_i/bij_{c_i}\}$ we have

1. $\sigma, \omega'' \models \bigwedge_c \forall z_c \exists! 1 \leq j \leq |bij_c| (bij_c : j = z_c)$
2. $\sigma, \omega'' \models \bigwedge_c \forall i(Act(i, State(m)) \leftrightarrow bij_c : i \neq nil)$
3. $\sigma, \omega'' \models \bigwedge_c \forall 1 \leq j \leq |bij_c| Bij(H^c, bij_c : j, j, State(m))$

Let f be an *osi* such that for an arbitrary c_i , $\alpha \in \sigma^{(c_i)}$ we have $\alpha_i([\beta]_{c_i}) = \alpha$ iff $f^{c_i}(\alpha) = \beta$. Note that this is well-defined because of 1 and 2.

By 3 and an application of 9.13 we have $f(h_{\sigma,\alpha}) = h_{\sigma_1, f^c(\alpha)}$, for an arbitrary c and $\alpha \in \sigma^{(c)}$. (Note that we have $Code(\omega, \sigma, f)$ and $f(\sigma) \approx \sigma_1$.)

Finally, applying theorem 9.9 gives us

$$(f^{-1}(X_0), f^{-1}(\sigma_0)) \rightarrow^{f^{-1}(h)} (f^{-1}(X_1), f^{-1}(\sigma_1)).$$

(Note that we have $Init_\rho((f^{-1}(X_0), f^{-1}(\sigma_0)))$ and that $f^{-1}(X_1)(\alpha)$ is normal, $\alpha \in f^{-1}(\sigma_1)^{(c)}$, c arbitrary.)

It then follows that for an arbitrary c , $\alpha \in \sigma^{(c)}$: $h_{\sigma,\alpha} = f^{-1}(h_{\sigma_1, f^c(\alpha)}) = h_{f^{-1}(\sigma_1), \alpha}$. (The second equality is an instance of the general fact that for an arbitrary σ , $\alpha \in \sigma^{(c)}$, and *osi* f we have $f(h_{\sigma,\alpha}) = h_{f(\sigma), f^c(\alpha)}$.)

The other way around: Suppose for the configurations $(X_0, \sigma_0), (X_1, \sigma_1)$ we have

- For some history $h : (X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$, where $Init_\rho((X_0, \sigma_0))$, and for $\alpha \in \sigma_0^{(c_n)}$ we have $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p^{c_n}$.
- For an arbitrary c and $\alpha \in \sigma_1^{(c)}$ we have that $X_1(\alpha)$ is normal.
- For an arbitrary c we have $\sigma^{(c)} = \sigma_1^{(c)}$, and for every $\alpha \in \sigma^{(c)}$ we have $h_{\sigma,\alpha} = h_{\sigma_1,\alpha}$.

Let $\gamma_1 = [(X_0, \sigma_0)]$, $\gamma_2 = [(X_1, \sigma_1)]$, and $\gamma_3 = [\alpha]_c$, where $\alpha \in \sigma_0^{c_n}$. Furthermore let $\omega' = \omega\{\gamma_1, \gamma_2, \gamma_3/n, m, k\}$.

By $(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$ and $Init_\rho((X_0, \sigma_0))$ we have $\sigma, \omega' \models Trans(n, m)$.

By $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p^{c_n}$ and lemma 9.3 we infer $\sigma, \omega' \models p'[u, v][State(n), k/u, v]$. Furthermore from $\alpha \in \sigma_0^{(c_n)}$ it follows that $\sigma, \omega' \models Act(k, State(n))$.

From the second clause above it immediately follows that $\sigma, \omega' \models \forall i (Act_c(i, m) \rightarrow Norm_c(i, m))$.

Finally, let α_i be a sequence of objects of class c_i such that for $\beta \in \sigma^{(c_i)}$ we have $\alpha_i([\beta]_{c_i}) = \beta$. So α_i stores every existing object of $\sigma^{(c_i)}$ at a position which equals its code number. Furthermore let $\omega'' = \omega'\{\alpha_i/bij_{c_i}\}_i$. It then follows that

$$\sigma, \omega'' \models \bigwedge_c \forall z_c \exists! 1 \leq j \leq |bij_c| (bij_c : j = z_c).$$

From $\sigma^{(c)} = \sigma_1^{(c)}$ we infer that

$$\sigma, \omega'' \models \bigwedge_c \forall i (Act(i, State(m)) \leftrightarrow bij_c : i \neq \text{nil}).$$

As $h_{\sigma, \alpha} = h_{\sigma_1, \alpha}$, for $\alpha \in \sigma^{(c)}$, c arbitrary, we have by lemma 9.13 that

$$\sigma, \omega'' \models \bigwedge_c \forall 1 \leq j \leq |bij_c| Bij(H^c, bij_c : j, j, State(m))$$

(take for the osi f the family of identity mappings). \square

9.4 Expressing preconditions and postconditions

We next show how to express $Pre(S)$ and $Post(S)$ as defined in the section on completeness. First we define some local assertions which partly express an isomorphism between a local state θ and some local state $\langle \alpha, \sigma(\alpha) \rangle$, for some σ and α , in terms of the code numbers for α and σ .

Definition 9.15

We define

- $LCode_{c_i}^c(x_{c_i}^c, y_{c_i}^c, n, m) =$
 $x_{c_i}^c \doteq y_{c_i}^c \leftrightarrow Val(n, [x_{c_i}^c], m) \doteq Val(n, [y_{c_i}^c], m)$
- $LCode_c^c(x_c^c, self, n, m) =$
 $x_c^c \doteq self \leftrightarrow Val(n, [x_c^c], m) \doteq n$
- $LCode_{int}^c(x_{int}^c, n, m) =$
 $x_{int}^c \doteq nil \rightarrow Val(n, [x_{int}^c], m) \doteq 0 \wedge$
 $x_{int}^c \neq nil \rightarrow Val(n, [x_{int}^c], m) \doteq [x_{int}^c]_{int}$
- $LCode_{int^*}^c(x_{int^*}^c, n, m) =$
 $\forall 1 \leq k \leq |x_{int^*}^c|$
 $(x_{int^*}^c : k \doteq nil \rightarrow Ell(Val(n, [x_{int^*}^c], m), [k]_{int}) \doteq 0 \wedge$
 $x_{int^*}^c : k \neq nil \rightarrow Ell(Val(n, [x_{int^*}^c], m), [k]_{int}) \doteq [x_{int^*}^c : k]_{int} \wedge$
 $|x_{int^*}^c| \doteq Len(Val(n, [x_{int^*}^c], m)))$
- $LCode_{c^*}^c(x_{c^*}^c, self, n, m) =$
 $\forall 1 \leq k \leq |x_{c^*}^c|$
 $(x_{c^*}^c : k \doteq self \leftrightarrow Ell(Val(n, [x_{c^*}^c], m), [k]_{int}) \doteq n)$
- $LCode_{c_i, c_i^*}^c(x_{c_i}^c, y_{c_i^*}^c, n, m) =$
 $\forall 1 \leq k \leq |y_{c_i^*}^c|$
 $(y_{c_i^*}^c : k \doteq x_{c_i}^c \leftrightarrow Ell(Val(n, [y_{c_i^*}^c], m), [k]_{int}) \doteq Val(n, [x_{c_i}^c], m))$

- $LCode_{c_i}^c(x_{c_i}^c, y_{c_i}^c, n, m) =$
 $\forall 1 \leq k \leq |x_{c_i}^c| \forall 1 \leq l \leq |y_{c_i}^c|$
 $(x_{c_i}^c : k \doteq y_{c_i}^c : l \leftrightarrow \text{Elt}(\text{Val}(n, [x_{c_i}^c], m), [k]_{\text{Int}}) \doteq \text{Elt}(\text{Val}(n, [y_{c_i}^c], m), [l]_{\text{Int}})) \wedge$
 $|x_{c_i}^c| \doteq \text{Len}(\text{Val}(n, [x_{c_i}^c], m)) \wedge |y_{c_i}^c| \doteq \text{Len}(\text{Val}(n, [y_{c_i}^c], m))$

To explain the interpretation of these assertions consider the assertion $LCode(x, y, n, m)$, where x and y are of a type $c \in C$. The idea is that the variable n is interpreted as the code of some object α and the variable m as the code of some state σ such that α exists in σ . The assertion $LCode(x, y, n, m)$ then states that in the current local state the variables x and y denote the same object iff they denote the same object in $\sigma(\alpha)$, i.e., $\sigma(\alpha)(x) = \sigma(\alpha)(y)$.

Given the above definition we now define a local assertion which describes an isomorphism between a local state θ and a local state $\langle \alpha, \sigma(\alpha) \rangle$ in terms of the codes for α and σ .

Definition 9.16

Let n and m be two distinct logical integer variables. We define $LCode(n, m)$ to be the following local assertion:

$$\begin{aligned} & Act_c(n, m) \wedge \\ & \bigwedge_{c'} \bigwedge_{x, y \in IVar_{c'}} LCode_{c'}(x, y, n, m) \wedge \bigwedge_{x \in IVar_c} LCode_c(x, \text{self}, n, m) \wedge \\ & \bigwedge_{x \in IVar_{\text{Int}}} LCode_{\text{Int}}(x, n, m) \wedge \bigwedge_{x \in IVar_{\text{Int}^*}} LCode_{\text{Int}^*}(x, n, m) \wedge \\ & \bigwedge_{c'} \bigwedge_{x, y \in IVar_{c'^*}} LCode_{c'^*}(x, y, n, m) \wedge \bigwedge_{x \in IVar_{c'^*}} LCode_{c'^*}(x, \text{self}, n, m) \wedge \\ & \bigwedge_{c'} \bigwedge_{x \in IVar_{c'}, y \in IVar_{c'^*}} LCode_{c', c'^*}(x, y, n, m) \end{aligned}$$

We have the following lemma about the meaning of the local assertion $LCode(n, m)$.

Lemma 9.17

For $\sigma, \alpha \in \sigma^{(c)}$, ω and θ^c we have

$$\begin{aligned} & \theta, \omega\{[\alpha]_c, [\sigma]/n, m\} \models LCode^c(n, m) \text{ iff} \\ & \text{there exists an } \text{osi } f \text{ such that } f(\theta^c) = \langle \alpha, \sigma(\alpha) \rangle. \end{aligned}$$

Proof

Let f be such that $f(\theta^c) = \langle \alpha, \sigma(\alpha) \rangle$. Now let x and y be two variables of type, say, c' . The other cases are treated similarly. We have

$$\theta \models x \doteq y \text{ iff } \langle \alpha, \sigma(\alpha) \rangle \models x \doteq y \text{ iff } \text{Val}([\alpha]_c, [x], [\sigma]) = \text{Val}([\alpha]_c, [y], [\sigma]).$$

Note that the first equivalence follows from theorem 9.8. On the other hand using $\theta^c, \omega\{[\alpha]_c, [\sigma]/n, m\} \models LCode^c(n, m)$ it is not difficult to construct an *osi* f such that for $\beta \in \mathbf{O}^{c'}$, c' arbitrary, we have: If β is referred to by some variable in θ we put $f^{c'}(\beta) = \gamma$, where γ is the object referred to by the same variable in $\langle \alpha, \sigma(\alpha) \rangle$. It then follows from the construction that $f(\theta) = \langle \alpha, \sigma(\alpha) \rangle$. \square

Finally we are ready to express $Pre(R)$ and $Post(R)$ in our assertion language.

Lemma 9.18

Let n, m, k be three distinct logical integer variables. We define

$$\begin{aligned} Pre(R) = & \\ \exists n, m, k \quad & (Trans(n, m) \wedge p'[u, v][State(n), k/u, v] \wedge Act_{c_n}(k, State(n)) \wedge \\ & \exists i(LCode(i, State(m)) \wedge Prog_c(i, m) = Before([R], [S^{(c)}])) \\ &) \end{aligned}$$

$Post(R)$ is defined analogously.

The assertion $\exists i(LCode(i, State(m)) \wedge Prog_c(i, m) = Before([R], [S^{(c)}]))$ will correspond to the phrase “for $\alpha \in \sigma_1^{(c_i)}$ we have $\langle \alpha, \sigma_1(\alpha) \rangle = \theta^{c_i}$ and $X_1(\alpha) = Before(R, S^{(c_i)})$ ”.

Proof

Let $\theta, \omega \models Pre(R)$. So there exist $\gamma_1, \gamma_2, \gamma_3$ such that $\theta, \omega' \models Pre(R)$, where $\omega' = \omega\{\gamma_1, \gamma_2, \gamma_3/n, m, k\}$.

By $\theta, \omega' \models Trans(n, m)$ we have that there exist configurations $(X_0, \sigma_0), (X_1, \sigma_1)$ such that $(X_0, \sigma_0) \rightarrow^h (X_1, \sigma_1)$ for some history h , $Init_{\rho'}((X_0, \sigma_0))$, and $[(X_0, \sigma_0)] = \gamma_1$, $[(X_1, \sigma_1)] = \gamma_2$.

From $\theta, \omega' \models p'[u, v][State(n), k/u, v] \wedge Act_{c_n}(k, State(n))$ by an application of lemma 9.3 it follows that $\langle \alpha, \sigma_0(\alpha) \rangle, \omega \models p'$, where $\alpha \in \sigma_0^{(c_n)}$.

Next let γ and $\omega'' = \omega'\{\gamma/i\}$ be such that $\theta, \omega'' \models LCode(i, State(m)) \wedge Prog_c(i, m) = Before([R], [S^{(c)}])$. It then follows, by an application of the previous lemma, that for $\alpha \in \sigma_1^{(c)}$ such that $[\alpha] = \gamma$ we have $X_1(\alpha) = Before(R, S^{(c)})$ and $f(\theta) = \langle \alpha, \sigma_1(\alpha) \rangle$ for some *osi* f .

Finally, an application of theorem 9.9 gives us

$$(f^{-1}(X_0), f^{-1}(\sigma_0)) \rightarrow^{f^{-1}(h)} (f^{-1}(X_1), f^{-1}(\sigma_1)).$$

Note that we now have $f^{-1}(X_1)(\theta(1)) = X_1(f^c(\theta(1))) = X_1(\alpha) = Before(R, S^{(c)})$ and $f^{-1}(\sigma_1)(\theta(1)) = f^{-1}(\sigma_1(f^c(\theta(1)))) = f^{-1}(\sigma_1(\alpha)) = \theta(2)$.

The other way around we invite the reader to check. \square

10 Conclusion

We have developed a proof system for the partial correctness of programs of a parallel language with dynamic process creation. The basic ingredients for dealing with parallelism in this proof system are the same as in a proof system for CSP [AFR], but they have been enhanced considerably to deal with the present, much more powerful programming language. One of the main problems we solved is how to reason about the dynamically evolving pointer structures that can arise during the execution of a program. A previous proof system [Bo] did this by considering pointers simply as numbers, which is of course not very abstract. Our present proof system allows reasoning at an abstraction level which is as high as that of the programming language, using a technique developed for a sequential language [Am1].

We have proved that the system is sound and complete. Again, the techniques developed for CSP [Ap2] could be used only after drastic modifications. Special care was necessary to be able to represent, in a finite number of variables, complete computations involving an unbounded number of objects. Also the programming language had to be extended to allow instance variables of sequence types. This extension, however, does not seem to be necessary in concrete proofs.

We have already mentioned that our language and our proof techniques can be considered as very powerful extensions to CSP [Ho2, AFR, Ap2]. A different kind of extension, where processes can split themselves recursively into subprocesses, is dealt with in [ZREB], for example. Here, the reasoning is not only based on *states*, as in our case, but also on *traces* (communication histories). The limitation of the possible process interconnection structures to trees allows a nice form of compositionality. In [Mel], a language similar to ours is tackled with trace-based reasoning. The problem of dynamic pointer structures is not dealt with explicitly.

We do not have the illusion that the proof system presented in this paper is suitable for proving 'practical' programs correct. One of the problems is that too much information is centralized in the global invariant. This seems unavoidable with the completely dynamic process structures allowed by our language. We would not like to dispense with this flexibility altogether, but we hope that by a judicious combination of our techniques with the compositional techniques developed elsewhere, future systems will allow us to have the best of both worlds.

Acknowledgements

We thank Jaco de Bakker, Arie de Bruin, Joost Kok, John-Jules Meyer, Jan Rutten and Erik de Vink, members of the Amsterdam Concurrency Group, for participating in discussions of preliminary versions of the proof system presented in this paper.

References

- [Am1] P.H.M. America: *A Proof Theory for a Sequential Version of POOL*. ESPRIT project 415A, Doc. No. 188, Philips Research Laboratories, Eindhoven, the Netherlands, October 1986.
- [Am2] P.H.M. America: *Issues in the Design of a Parallel Object-Oriented Language*. ESPRIT project 415A, Doc. No. 452, Philips Research Laboratories, Eindhoven, the Netherlands, November 1988. To appear in *Formal Aspects of Computing*.
- [Am3] P.H.M. America: *A Behavioural Approach to Subtyping in Object-Oriented Programming Languages*. Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages, Viareggio, Italy, February 6-8, 1989 Also appeared in *Philips Journal of Research*, Vol. 44, No. 2/3, July 1989, pp. 365-383.
- [AFR] K.R. Apt, N. Francez, W.P. de Roever: *A proof system for Communicating Sequential Processes*, *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3, July 1980, pp. 359-385.
- [Ap1] K.R. Apt: *Ten years of Hoare logic: a survey — part I*. *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp. 431-483.
- [Ap2] K.R. Apt: *Formal justification of a proof system for Communicating Sequential Processes*. *Journal of the ACM*, Vol. 30, No. 1, January 1983, pp. 197-216.
- [Ba] J.W. de Bakker: *Mathematical theory of program correctness*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.
- [Bo] F.S. de Boer: *A proof rule for process creation*. M. Wirsing (ed.): *Formal Description of Programming Concepts*. Proceedings of the third IFIP WG 2.2 working conference, Gl. Avernæs, Ebberup, Denmark, August 25-28, 1986, North-Holland.
- [GR] A. Goldberg and D. Robson: *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, 1984.
- [Ho1] C.A.R. Hoare: *An axiomatic basis for computer programming*. *Communications of the ACM*, Vol. 12, No. 10, 1969, pp. 567-580, 583.
- [Ho2] C.A.R. Hoare: *Communicating Sequential Processes*. *Communications of the ACM*, Vol. 21, No. 8, 1978, pp. 666-677.
- [HR] J. Hooman, W.P. de Roever: *The quest goes on: towards compositional proof systems for CSP*. J.W. de Bakker, W.P. de Roever, G. Rozenberg (eds.): *Current Trends in Concurrency*, Springer LNCS 224, 1986, pp. 343-395.

- [Mel] S. Meldal: *Axiomatic Semantics of Access Type Tasks in Ada*. Report No. 100, Institute of Informatics, University of Oslo, Norway, May 1986. To appear in Distributed Computing.
- [Mey] B. Meyer: *Object-Oriented Software Construction*. Prentice-Hall, 1988.
- [TZ] J.V. Tucker, J.I. Zucker: *Program Correctness over Abstract Data Types, with Error-State Semantics*, CWI Monographs 6, North-Holland, 1988.
- [ZREB] J. Zwiers, W.P. de Roever, P. van Emde Boas: *Compositionality and concurrent networks: soundness and completeness of a proof system*. In Proceedings of the 12th ICALP, Nafplion, Greece, July 15–19, 1985, Springer LNCS 194, pp. 509–519.

A Index of notation

A.1 Sets and their typical elements

Typ. elt.	Set	Description	Where defined
i, j, k, n	\mathbf{Z}	integers	-
c	C	class names	above definition 2.1
d	C^+	data types	above definition 2.1
x_d^c	$IVar_d^c$	instance variables of type d in class c	above definition 2.1
e_d^c	Exp_d^c	expressions of type d in class c	definition 2.1
S^c	$Stat^c$	statements in class c	definition 2.2
ρ	$Prog$	programs	definition 2.3
l_d^c	$LExp_d^c$	local expressions of type d in class c	definition 3.1
p^c	$LAss^c$	local assertions in class c	definition 3.2
d^*	C^*	sequence types	above definition 3.3
a	C^\dagger	data and sequence types	above definition 3.3
z_a	$LogVar_a$	logical variables of type a	above definition 3.3
g_a	$GExp_a$	global expressions of type a	definition 3.3
P	$GAss$	global assertions	definition 3.4

A.2 Syntactic transformations

Notation	Where defined
$l_d^c \downarrow g_c$	definition 3.5
$p^c \downarrow g_c$	definition 3.5
$[e/x]$	standard
$[g/z.x]$	definition 4.3
$[\text{new}/z]$	definitions 4.5 and 4.6
$[z_{\text{Bool}^\bullet}, z_c/z_c^\bullet]$	definition 4.7

A proof theory for the language POOL

Frank de Boer

Contents

1	Introduction	202
2	The programming language	205
2.1	The syntax	206
3	The assertion language	209
3.1	The local assertion language	209
3.2	The global assertion language	211
3.3	Correctness formulas	213
4	The proof system	217
4.1	The local proof system	217
4.2	The intermediate proof system	219
4.3	The global proof system	227
5	Semantics	232
5.1	Semantics of the assertion languages	233
5.2	The transition system	236
5.3	Truth of correctness formulas	242

6 Soundness	243
6.1 The local proof system	243
6.2 The intermediate proof system	244
6.3 The global proof system	248
7 Completeness	253
8 Conclusion	267
References	267

1 Introduction

The goal of this paper is to develop a formal system for reasoning about the correctness of a certain class of parallel programs. We shall consider programs written in the language POOL, a parallel object-oriented language [Am]. POOL makes use of the structuring mechanisms of object-oriented programming, integrated with the concepts for expressing concurrency: processes and rendezvous.

A program of the language POOL describes the behaviour of a whole system in terms of its constituents, *objects*. These objects have the following important properties: First of all, each object has an independent activity of its own: a process that proceeds in parallel with all the other objects in the system. Second, new objects can be created at any point in the program. The identity of such a new object is at first only known to itself and its creator, but from there it can be passed on to other objects in the system. Note that this also means that the number of processes executing in parallel may increase during the evolution of the system.

Objects possess some internal data, which they store in *variables*. The value of a variable is either an element of a predefined data type (Int or Bool), or it is a *reference* to another object. The variables of one object are not accessible to other objects. The objects can interact only by sending *messages*. A message is transferred synchronously from the sender to the receiver, i.e., sending and receiving a message take place at the same time. A message contains a *method* name (procedures are called methods in POOL) and a sequence of actual parameters specified by the sender of the message. While the receiver of the message is executing the method the execution of the sender

is suspended. When the result of the execution of the method has been received by the sender of the message it resumes its activity.

Thus we see that a system described by a program in the language POOL consists of a dynamically evolving collection of objects, which are all executing in parallel, and which know each other by maintaining and passing around references. This means that also the communication structure of the processes is completely dynamic, without any regular structure imposed on it a priori. This is to be contrasted with the static structure (a fixed number of processes, communicating with statically determined partners) in [AFR] and the tree-like structure in [ZREB].

One of the main proof theoretical problems of such an object-oriented language is how to reason about dynamically evolving *pointer structures*. We want to reason about these structures on an abstraction level that is *at least as high as that of the programming language*. In more detail, this means the following:

- The only operations on “pointers” (references to objects) are
 - testing for equality
 - dereferencing (looking at the value of an instance variable of the referenced object)
- In a given state of the system, it is only possible to mention the objects that exist in that state. Objects that do not (yet) exist never play a role.

Strictly speaking, direct dereferencing is not even allowed in the programming language, because each object only has access to its own instance variables. However, for the time being we allow it in the assertion language. Otherwise, even more advanced techniques would be necessary to reason about the correctness of a program.

The above restrictions have quite severe consequences for the proof system. The limited set of operations on pointers implies that first-order logic is too weak to express some interesting properties of pointer structures. Therefore we have to extend our assertion language to make it more expressive. We will do so by allowing the assertion language to reason about *finite sequences* of objects. Furthermore we have to define some special substitution operations to model aliasing and the creation of new objects.

To deal with parallelism, the proof theory we shall develop uses the concepts of *cooperation test*, *global invariant*, *bracketed section* and *auxiliary variables*. These concepts have been developed in the proof theory of CSP [AFR], and have been applied to quite a variety of concurrent programming languages [HR]. In fact our proof method generalizes the application of these concepts to the language Ada [GR]. The main difference between the ADA-rendezvous and the rendezvous mechanism of POOL

consists in that in POOL we have *no* static bound to the recursion depth of the rendezvous mechanism whereas in ADA there is. A consequence of this is that the proof method for ADA is *incomplete* when applied to the language POOL. Here completeness means that every true property of a program that can be expressed in the assertion language used can also be proved formally in the proof system. This incompleteness can be resolved by some additional reasoning mechanism which essentially formalizes reasoning about *invariance* properties of a rendezvous.

Described very briefly this proof method applied to our language consists of the following elements:

- A *local* stage. Here we deal with all statements that do not involve message passing or object creation. These statements are proved correct with respect to pre- and postconditions formulated in a *local assertion language*, which only talks about the current object in isolation. At this stage, we use the *assumption/commitment* formalism [HR] to describe the behaviour of the rendezvous and creation statements. The assertions describing the assumptions and commitments are formulated in the local assertion language. The assumptions about these statements then will be verified in the next stage.
- An *intermediate* stage. In this stage the above assumptions about rendezvous and creation statements are verified. Here a *global assertion language*, which reasons about all the objects in the system, is used. For each creation statement and for each pair of possibly communicating rendezvous statements it is verified that the specification used in the local proof system is consistent with the global behaviour.
- A *global* stage. Here some properties of the system as a whole can be derived from a kind of standard specification that arises from the intermediate stage. Again the global assertion language is used.

We have proved that the proof system is sound and complete with respect to a formal semantics. Soundness means that everything that can be proved using the proof system is indeed true in the semantics. Due to the abstraction level of the assertion language we had to modify considerably the standard techniques for proving completeness ([Ap]). In the completeness proof we combine the techniques for proving completeness of the proof system for recursive procedures of sequential languages ([Ap2]) based upon the expressibility of the strongest postcondition with the techniques of [Ap] developed for CSP.

Our paper is organized as follows: In the following section we describe the programming language POOL. In section 3 we define two assertion languages, one to describe the internal data of an object and one to describe a complete system of objects. Also

in section 3 we show how to specify the behaviour of an object and a system of objects. In section 4 we describe the proof system. The semantics of the programming language, the assertion languages, and the specification languages are given in section 5. The soundness and completeness of the proof system are treated in the sections 6 and 7.

2 The programming language

In this section we define a abstract version of the programming language POOL of which we shall study the proof theory.

The most important concept is the concept of an *object*. This is an entity containing data and procedures (*methods*) acting on these data. Furthermore, every object has an internal activity of its own. The data are stored in *variables*, which come in two kinds: *instance variables*, whose lifetime is the same as that of the object they belong to, and *temporary variables*, which are local to a method and last as long as the method is active. Variables can contain references to other objects in the system (or even the object under consideration itself). The object a variable refers to (its *value*) can be changed by an *assignment*. The value of a variable can also be nil, which means that it refers to no object at all.

The variables of an object cannot be accessed directly by other objects. The only way for objects to interact is by sending *messages* to each other. If an object sends a message, it specifies the receiver, a method name, and possibly some parameter objects. The reception of a message is modeled by means of an *answer* statement which specifies some method names for which an incoming message can be answered. When an object executing an answer statement receives a message for one of the specified methods it starts to execute the particular method, using the parameters in the message. Note that this method can, of course, access the instance variables of the receiver. The method returns a result, an object, which is sent back to the sender. The sender of a message is *blocked* until the result comes back, that is, it cannot answer any message while it still has an outstanding message of its own. Therefore, when an object sends a message to itself (directly or indirectly) this will lead to abnormal termination of the program. This is an important difference with some other object-oriented languages, like Smalltalk-80 [Go]. After the result has been sent back both the sender and the receiver resume their own activities.

Objects are grouped into *classes*. Objects in one class (the *instances* of the class) share the same methods and the same statement which specifies their internal activity, so in a certain sense they share the same behaviour. New instances of a given class can be created at any time. There are two standard classes, *Int* and *Bool*, of integers and booleans, respectively. They differ from the other classes in that their instances

already exist at the beginning of the execution of the program and no new ones can be created. Moreover, some standard operations on these classes are defined.

A program essentially consists of a number of class definitions, together with a statement which specifies the behaviour of the *root-object*, the object which starts the execution. So initially only this object exists: the others still have to be created.

2.1 The syntax

In order to describe the language POOL, which is strongly typed, we use *typed* versions of all variables, expressions, etc. These types are indicated by subscripts or superscripts in this language description. Often, when this typing information is redundant, it is omitted. Of course, for a practical version of the language, a syntactical variant, in which the type of each variable is indicated by a *declaration*, is easier to use.

Assumption 2.1

We assume the following sets to be given:

- A set C of *class names*, with typical element c (this means that metavariables like c, c', c_1, \dots range over elements of the set C . We assume that $\text{Int}, \text{Bool} \notin C$ and define the set $C^+ = C \cup \{\text{Int}, \text{Bool}\}$ with typical element d .
- For each $c \in C$ and $d \in C^+$ we assume a set $IVar_d^c$ of *instance variables* of type d in class c . By this we mean that such a variable may occur in the definition of class c and that its contents will be an object of type d . The set $IVar_d^c$ will have as a typical element x_d^c . We define $IVar = \bigcup_{c,d} IVar_d^c$ and $IVar^c = \bigcup_d IVar_d^c$.
- For each $d \in C$ we assume a set $TVar_d$ of *temporary variables* of type d , with typical element u_d . We define $TVar = \bigcup_d TVar_d$ and $ITvar = IVar \cup TVar$.
- We shall let the metavariable n range over elements of \mathbb{Z} , the set of whole numbers.
- For each $c \in C$ and $d_0, \dots, d_n \in C^+$ ($n \geq 0$) we assume a set $MName_{d_0, \dots, d_n}^c$ of *method names* of class c with result type d_0 and parameter types d_1, \dots, d_n . The set $MName_{d_0, \dots, d_n}^c$ will have m_{d_0, \dots, d_n}^c as a typical element.

Now we can specify the syntax of our language. We start with the expressions:

Definition 2.2

For any $c \in C$ and $d \in C^+$ we define the set Exp_d^c of *expressions* of type d in class c , with typical element e_d^c , as follows:

$$\begin{aligned}
e_d^c ::= & x_d^c \\
& | u_d \\
& | \text{nil}_d \\
& | \text{self} && \text{if } c = d \\
& | \text{true} \mid \text{false} && \text{if } d = \text{Bool} \\
& | n && \text{if } d = \text{Int} \\
& | e_{1_{d'}}^c \dot{=} e_{2_{d'}}^c && \text{if } d = \text{Bool} \\
& | e_{1_{\text{Int}}}^c + e_{2_{\text{Int}}}^c && \text{if } d = \text{Int} \\
& \vdots \\
& | e_{1_{\text{Int}}}^c < e_{2_{\text{Int}}}^c && \text{if } d = \text{Bool} \\
& \vdots
\end{aligned}$$

The expression `self` denotes the current object. The expression `nil` stands for “undefined”. The intuitive meaning of the other expressions will probably be clear. Note that in the language we put a dot over the equal sign ($\dot{=}$) to distinguish it from the equality sign we use in the meta-language.

Definition 2.3

The set $SExp_d^c$ of expressions with possible *side effect* of type d in class c , with typical element s_d^c , is defined as follows:

$$\begin{aligned}
s_d^c ::= & e_d^c \\
& | \text{new}_d && \text{if } d \in C \ (d \neq \text{Int}, \text{Bool}) \\
& | e_{0_{c_0}}^c ! m_{d,d_1,\dots,d_n}^{c_0} (e_{1_{d_1}}^c, \dots, e_{n_{d_n}}^c) \ (n \geq 1) \ e_1 = \text{self}
\end{aligned}$$

The first kind of side effect expression is a normal expression, which has no actual side effect, of course. The second kind is the creation of a new object. This new object will also be the value of the side effect expression. The third kind of side effect expression specifies that a message is to be sent to the object that results from e_0 , with method name m and with arguments (the objects resulting from) e_1, \dots, e_n . Note that we require the first argument to be the sender itself (so we have that $d_1 = c$). This requirement is not present in the language POOL. It is introduced for proof theoretical reasons only. However, every POOL program can be transformed into an equivalent one satisfying this requirement.

Definition 2.4

The set $Stat^c$ of *statements* in class c , with typical element S^c , is defined by:

$$\begin{array}{lcl}
S^c & ::= & x_d^c \leftarrow s_d^c \\
& | & u_d \leftarrow s_d^c \\
& | & \text{answer}(m_1, \dots, m_n) \\
& | & S_1^c; S_2^c \\
& | & \text{if } e_{\text{Bool}}^c \text{ then } S_1^c \text{ else } S_2^c \text{ fi} \\
& | & \text{while } e_{\text{Bool}}^c \text{ do } S^c \text{ od}
\end{array}$$

The execution of an answer statement $\text{answer}(m_1, \dots, m_n)$ consists of waiting for a message for one of the methods m_1, \dots, m_n . When such a message arrives the corresponding method is executed. In case of the arrival of several messages one is chosen non-deterministically. The intuitive meaning of the other statements will probably be clear.

Definition 2.5

The set $\text{MethDef}_{d_0, \dots, d_n}^c$ of *method definitions* of class c with result type d_0 and parameter types d_1, \dots, d_n (with typical element μ_{d_0, \dots, d_n}^c) is defined by:

$$\mu_{d_0, \dots, d_n}^c ::= (u_{d_1}, \dots, u_{d_n}) : S^c \uparrow e_{d_0}^c$$

Here we require that the u_{d_i} are all different and that none of them occurs at the left hand side of an assignment in S^c (and that $n \geq 1$).

When an object is sent a message, the method named in the message is invoked as follows: The variables u_1, \dots, u_n (the parameters of the methods) are given the values specified in the message, all other temporary variables are initialized to nil, and then the statement S is executed. After that the expression e is evaluated and its value, the result of the method, is sent back to the sender of the message, where it will be the value of the send-expression that sent the message.

Definition 2.6

The set $\text{ClassDef}_{m_1, \dots, m_n}^c$ of *class definitions* of class c defining methods m_1, \dots, m_n , with typical element D_{m_1, \dots, m_n}^c , is defined by:

$$D_{m_1, \dots, m_n}^c ::= \langle m_{d_1}^c \Leftarrow \mu_{d_1}^c, \dots, m_{d_n}^c \Leftarrow \mu_{d_n}^c \rangle : S^c$$

where we require that all the method names are different (and $n \geq 0$) and $TVar(S^c) = \emptyset$. (Here $TVar(S^c)$ denotes the set of temporary variables occurring in S^c . Furthermore, \bar{d}_i , denotes a sequence of types.)

Definition 2.7

The set $\text{Unit}_{m_1, \dots, m_k}^{c_1, \dots, c_n}$ of *units* with classes c_1, \dots, c_n defining methods m_1, \dots, m_k , with typical element $U_{m_1, \dots, m_k}^{c_1, \dots, c_n}$, is defined by:

$$U_{m_1, \dots, m_k}^{c_1, \dots, c_n} ::= D_{1\tilde{m}_1}^{c_1}, \dots, D_{n\tilde{m}_n}^{c_n}$$

where $m_1, \dots, m_k = \tilde{m}_1, \dots, \tilde{m}_n$, that is, m_1, \dots, m_k results from concatenating the sequences of method names \tilde{m}_i . We require that all the class names are different.

Definition 2.8

Finally, the set $Prog^c$ of *programs* in class c , with typical element ρ^c , is defined by:

$$\rho^c ::= \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_n} | c : S^c \rangle$$

Here we require that c does not occur in c_1, \dots, c_n and that no assignment $x \leftarrow \text{new}$, x of type c , and $u \leftarrow \text{new}$, u of type c , occurs in $U_{m_1, \dots, m_k}^{c_1, \dots, c_n}$ and S^c . Finally, we require that $TVar(S^c) = \emptyset$. (The symbol ' $|$ ' is part of the syntax, not of the meta-syntax.)

We call a program $\rho = \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_n} | c : S^c \rangle$ *closed* iff every method name occurring in it is defined by U , and only variables x_d, u_d , with $d \in \{c_1, \dots, c_n, \text{Int}, \text{Bool}\}$, occur in ρ .

The interpretation of such a program is that the statement S is executed by some object of class c (the root object) in the context of the declarations contained in the unit U . We assume that at the beginning of the execution this root object is the only existing non-standard object. The additional requirement ensures that throughout the execution the root-object will be the only existing object of its own class.

3 The assertion language

In this section we define two different assertion languages, i.e., sets of assertions. An *assertion* is used to describe the state of (a part of) the system at one specific point during its execution. The first assertion language describes the *internal state* of a single object. This language will be called the *local* assertion language. The other one is to be used to describe a whole system of objects. The latter language will be called the *global* assertion language.

3.1 The local assertion language

First we introduce a new kind of variables: For $d = \text{Int}, \text{Bool}$, let $LVar_d$ be an infinite set of *logical variables* of type d , with typical element z_d . We assume that these sets are disjoint from the other sets of syntactic entities. Logical variables do not occur in a program, but only in assertions.

Definition 3.1

The set $LExp_d^c$ of *local expressions* of type d in class c , with typical element l_d^c , is

defined as follows:

$$\begin{aligned}
 l_d^c &::= z_d \\
 &| x_d^c \\
 &| u_d \\
 &| \text{self} \quad \text{if } d = c \\
 &| \text{nil} \\
 &| n \quad \text{if } d = \text{Int} \\
 &| \text{true} \mid \text{false} \quad \text{if } d = \text{Bool} \\
 &| l_{1\text{Int}}^c + l_{2\text{Int}}^c \quad \text{if } d = \text{Int} \\
 &\vdots \\
 &| l_{1d}^c \doteq l_{2d}^c \quad \text{if } d = \text{Bool}
 \end{aligned}$$

Definition 3.2

The set $LAss^c$ of *local assertions* in class c , with typical element p^c , is defined as follows:

$$\begin{aligned}
 p^c &::= l_{\text{Bool}}^c \\
 &| \neg p^c \\
 &| p_1^c \wedge p_2^c \\
 &\vdots \\
 &| \exists z_d p^c \quad d \in \{\text{Int}, \text{Bool}\}
 \end{aligned}$$

Local expressions l_d^c and local assertions p^c are evaluated with respect to the local state of an object of class c (plus a logical environment that assigns values to the logical variables). They talk about this single object in isolation. It is important to note that we allow only logical variables ranging over integers and booleans to occur in local expressions. The intuition behind this is that the internal data of an object consists of the objects stored in its instance variables, so only these objects and the standard ones are known by this object. A logical variable of a type $c' \in C$ would provide a “window” to the external world. Furthermore, as we will explain below, we will define the range of quantification over a class c' to be the set of *existing* objects of this class. But the set of existing objects of the class c' is a global aspect of the system, what is known locally is in general a subset of this set. So quantification over objects of some class c' can not be evaluated by looking only at the local state of some object.

3.2 The global assertion language

Next we define the global assertion language. As we want to quantify in the global assertion language also over objects of some arbitrary class c we now need for every $c \in C$ a new set $LVar_c$ of logical variables of type c , with typical element z_c . To be able to describe interesting properties of pointer structures we also introduce logical variables ranging over *finite sequences* of objects. To do so we first introduce for every $d \in C^+$ the type d^* of finite sequences of objects of type d . We define $C^* = \{d^* : d \in C^+\}$ and take $C^\dagger = C^+ \cup C^*$, with typical element a . Now we assume in addition for every $d \in C^+$ the set $LVar_{d^*}$ of logical variables of type d^* , which range over finite sequences of elements of type d . Therefore in total we now have a set $LVar_a$ of logical variables of type a for every $a \in C^\dagger$.

Definition 3.3

The set $GExp_a$ of *global expressions* of type a , with typical element g_a , is defined as follows:

$g_a ::=$	z_a	
	nil	
	n	if $a = \text{Int}$
	$true \mid false$	if $a = \text{Bool}$
	$g_c.x_d^c$	if $a = d$
	$g_{d^*} : g_{\text{Int}}$	if $a = d$
	$ g_{d^*} $	if $a = \text{Int}$
	$g_{1_{\text{Int}}} + g_{2_{\text{Int}}}$	if $a = \text{Int}$
	\vdots	
	if g_{Bool} then g_{1_a} else g_{2_a} fi	
	$g_{1_d} \doteq g_{2_d}$	if $a = \text{Bool}$

A global expression is evaluated with respect to a complete system of objects plus a logical environment. A complete system of objects consists of a set of existing objects together with their local states. The expression $g.x$ denotes the value of the variable x of the object denoted by g . Note that in this global assertion language we must explicitly specify the object of which we want to access the internal data. The expression $g_1 : g_2$ denotes the n^{th} element of the sequence denoted by g_1 , where n is the value of g_2 . (If the value of g_2 is less than 1 or greater than the length of the sequence denoted by g_1 we define the value of $g_1 : g_2$ to be undefined, i.e., equivalent to nil .) The expression $|g|$ denotes the length of the sequence denoted by g . For sequence types the expression nil denotes the empty sequence. The conditional expression if-then-else-fi is introduced to facilitate the handling of aliasing. If the condition is undefined, i.e.,

equals nil, then the result of the conditional expression is undefined, too. Finally, note that we do not have temporary variables in the global assertion language, since objects that are not executing a method do not have temporary variables.

Definition 3.4

The set GA_{ss} of *global assertions*, with typical element P , is defined as follows:

$$\begin{aligned} P ::= & \text{gBool} \\ & | \neg P \mid P_1 \wedge P_2 \\ & \vdots \\ & | \exists z_a p \end{aligned}$$

Quantification over (sequences of) integers and booleans is interpreted as usual. However, quantification over (sequences of) objects of some class c is interpreted as ranging only over the *existing* objects of that class, i.e., the objects that have been created up to the current point in the execution of the program. For example, the assertion $\exists z_c \text{true}$ is false in some state iff there are no objects of class c in this state.

Next we define a transformation of a local expression or assertion to a global one. This transformation will be used to verify the assumptions made in the local proof system about the send, answer, and new-statements. These assumptions are formulated in the local language. As the reasoning in the cooperation test uses the global assertion language we have to transform these assumptions from the local language to the global one.

Definition 3.5

Given a local expression l_a^c , which does *not* contain temporary variables, we define $l_a^c[g_c/\text{self}]$ by induction on the complexity of the local expression l_a^c :

$$\begin{aligned} z_d[g_c/\text{self}] &= z_d \\ x_d^c[g_c/\text{self}] &= g_c.x_d^c \\ \text{self}[g_c/\text{self}] &= g_c \\ &\vdots \end{aligned}$$

The omitted cases follow directly from the transformation of the subexpressions. For a local assertion p^c , which does not contain temporary variables, we define $p^c[g_c/\text{self}]$ as follows:

$$\begin{aligned} l_{\text{Bool}}^c[g_c/\text{self}] &\text{ as above} \\ (\neg p^c)[g_c/\text{self}] &= (\neg p^c[g_c/\text{self}]) \\ &\vdots \\ (\exists z_d p^c)[g_c/\text{self}] &= \exists z_d (p^c[g_c/\text{self}]) \end{aligned}$$

The global assertion $p^c[g_c/\text{self}]$ expresses that the local assertion p^c is true for the object denoted by g_c . Note that we do not allow temporary variables to occur in p^c because they do not exist in the global assertion language.

3.3 Correctness formulas

In this section we define how we specify an object and a complete system of objects. For the specification of an object we use the assumption/commitment formalism ([HR]). First we introduce two sets of labels Lab_A and Lab_C such that $Lab_A \cap Lab_C = \emptyset$. Elements of $Lab_A \cup Lab_C$ we denote by l, \dots . We extend the class of statements by the rule

$$S ::= l$$

The execution of a label is equivalent to a skip statement. A label is used to mark a control point. We use labeled local assertions, notation: $l.p$, to characterize the state during the execution of the corresponding label. We can now give the definition of a specification of an object.

Definition 3.6

We define a *local correctness formula* to be of the following form:

$$(A, C : \{p^c\}S^c\{q^c\})$$

where

- $A \subseteq \{l.p^c : l \in Lab_A\}$
- $C \subseteq \{l.p^c : l \in Lab_C\}$.

Furthermore, we require that every label of the set A (C) occurs at most once.

The meaning of such a correctness formula is described informally as follows:

For an arbitrary prefix of a computation of S^c by an object of class c the following holds:

If

p^c holds initially and in an arbitrary state of this sequence whenever the object is executing a label $l \in Lab_A$ the corresponding assertion holds

then

if the object is about to execute a label $l \in Lab_C$ then the corresponding assertion holds and if the execution is terminated then q^c characterizes its final state.

A local correctness formula $(A, C : \{p\}S\{q\})$ formalizes reasoning about the local correctness of an object relative to assumptions concerning those parts of its local process that depend on the environment. This can be explained as follows: let, for example, R be a send statement occurring in S . An assumption about R is a Hoare-triple $\{p'\}R\{q'\}$, which states that whenever R is executed in a state satisfying p' then the resulting state will satisfy q' . Note that q' thus “quesses” the result of R , i.e., the value sent back. The correctness of a specification $\{p\}S\{q\}$ of S with respect to the assumption $\{p'\}R\{q'\}$ about R then amounts to the derivability of $\{p\}S\{q\}$ from $\{p'\}R\{q'\}$ using the standard Hoare-style proof system for sequential programs ([Ba]). Now this is equivalent to requiring that the *partial proof-outline* of S which consists of associating with S the precondition p and the postcondition q and with R the precondition p' and the postcondition q' can be extended to a *complete proof-outline* which associates with every substatement of S a precondition and a postcondition. In our framework the above mentioned partial proof-outline corresponds to the local correctness formula $(\{l.q'\}, \{l'.p'\} : \{p\}S'\{q\})$, where S' is obtained from S by replacing R by $l; R; l'$. In the following section we define a proof system for reasoning about local correctness formulas. The derivability from this system of the correctness formula $(\{l.q'\}, \{l'.p'\} : \{p\}S'\{q\})$ then amounts essentially to proving the derivability of $\{p\}S\{q\}$ from $\{p'\}R\{q'\}$ using the usual proof system for Hoare-triples.

The parts of a local process which depend on the environment are called *bracketed sections*:

Definition 3.7

Let R_1 and R_2 be statements in which there occur no temporary variables, and no send, answer, and new-statements. A bracketed section is a construct of one of the following forms:

- $l_1; R_1; l; x \leftarrow e_0!m(e_1, \dots, e_n); R_2; l_2$,
where $\text{Mod}(R_1) \cap \text{Var}(e_0, e_1, \dots, e_n) = \emptyset$.
- $l_1; R_1; x \leftarrow \text{new}; R_2; l_2$,
where $x \notin \text{Mod}(R_2)$.
- $l_1; \text{answer}(m_1, \dots, m_n); l_2$
- $m \leftarrow R_1; l_1; S; l_2; R_2 \uparrow e$,
where $\text{Mod}(R_2) \cap \text{Var}(e) = \emptyset$.

Here $\text{Mod}(R)$ is defined inductively as follows:

- $\text{Mod}(x \leftarrow s) = \{x\}$

- $Mod(answer(m_1, \dots, m_n)) = \bigcup_i Mod(S_i) \cap IVar$,
where S_i is the body of m_i
- $Mod(S_1; S_2) = Mod(S_1) \cup Mod(S_2)$
- $Mod(\text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}) = Mod(S_1) \cup Mod(S_2)$
- $Mod(\text{while } e \text{ do } S \text{ od}) = Mod(S)$

When m is declared as $R_1; l_1; S; l_2; R_2 \uparrow e$ we call R_1 its *prelude* and R_2 its *postlude*. Also we call the statement R_1 the prelude of the bracketed section $l; R_1; R; R_2; l'$ (R a new or a send statement) and R_2 its postlude. The restriction $Mod(R_1) \cap Var(e_0, \dots, e_n) = \emptyset$ of the first clause is introduced to ensure that the execution of R_1 does not affect the values of the expressions e_0, \dots, e_n , so that before the execution of R_1 we in fact know the object to which the request is made and the actual parameters. The restriction $Mod(R_2) \cap Var(e) = \emptyset$ of the last clause ensures that the execution of R_2 does not affect the result. Both restrictions will be used in the definition of the cooperation test.

In the following section we define a proof system for reasoning about local correctness formulas. The derivability from this system of a correctness formula $(A, C : \{p\}S\{q\})$ then amounts essentially to proving

$$\mathcal{A} \vdash \{p\}S'\{q\}$$

where $\mathcal{A} = \{\{C(l)\}R\{A(l')\} : l; R; l' \text{ a bracketed section occurring in } S\}$, and S' results from S by removing all labels, using the usual proof system for sequential programs. Here, given a set of labeled assertions X such that with each label occurring in X there corresponds at most one assertion, and a label l , we define

$$\begin{aligned} X(l) &= p && \text{if } l.p \in X \\ &= \text{true} && \text{otherwise.} \end{aligned}$$

However, with respect to the soundness proofs, correctness formulas $(A, C : \{p\}S\{q\})$, as they essentially represent a partial proof-outline of the version of S without labels, are more convenient.

Next we define intermediate correctness formulas, which describe the behaviour of objects executing a local statement (that is, a statement not involving any new, answer, or send statements), or a bracketed section containing a new-statement, from a global point of view.

Definition 3.8

An *intermediate correctness formula* can have one of the following two forms:

- $\{P\}(z_c, R^c)\{Q\}$, where R^c is a local statement containing no temporary variables or a bracketed section containing a new-statement. (A local statement is a statement containing no new, send, or answer statements).
- $\{P\}(z_{c_1}, R_1^{c_1}) \parallel (z'_{c_2}, R_2^{c_2})\{Q\}$, where z_{c_1} and z'_{c_2} are distinct logical variables and $R_1^{c_1}$ and $R_2^{c_2}$ are local statements.

The logical variables z_c , z_{c_1} and z'_{c_2} in the above constructs denote the objects that are considered to be executing the corresponding statements. More precisely, the meaning of the intermediate correctness formula $\{P\}(z, R)\{Q\}$ is as follows:

Every terminating execution of R by the object denoted by the logical variable z starting in a state satisfying P ends in a state satisfying Q .

The meaning of the second form of intermediate correctness formula, $\{P\}(z, R_1) \parallel (z', R_2)\{Q\}$, can be described as follows:

Every terminating parallel execution of R_1 by the object denoted by the logical variable z and of R_2 by the object denoted by z' starting in a state satisfying P will end in a state satisfying Q .

In the cooperation test a correctness formula $\{P\}(z, R)\{Q\}$, R a bracketed section containing a new-statement, will be used to justify the assumption associated with R . A correctness formula $\{P\}(z_1, R_1) \parallel (z_2, R_2)\{Q\}$, with R_1 the prelude of a bracketed section containing a send-statement, and R_2 a prelude of an answer-statement, will be used to justify the assumption about the parameters. Information about the actual parameters will be coded in P . On the other hand a correctness formula $\{P\}(z_1, R_1) \parallel (z_2, R_2)\{Q\}$, with R_1 the postlude of a bracketed section containing a send-statement, and R_2 a postlude of an answer-statement, will be used to justify the assumption about the result value and the assumption about the state after the execution of the answer-statement. Information about the result will be coded in P .

Finally, we have *global correctness formulas*, which describe a complete system:

Definition 3.9

A global correctness formula is of the form

$$\{p^c[z_c/\text{self}]\}\rho\{Q\}$$

where ρ is a program and c is the root class in ρ , and $TVar(p) = \emptyset$.

The variable z_c in such a global correctness formula denotes the root object. Initially this root object is the only existing object, so it is sufficient for the precondition of a complete system to describe only its local state. We obtain such a precondition by transforming some local assertion p^c to a global one. On the other hand, the final state of an execution of a complete system is described an arbitrary global assertion. The meaning of the global correctness formula $\{p[z/self]\}\rho\{Q\}$ can be rendered as follows:

If the execution of the unit ρ starts with a root object denoted by z that satisfies the local assertion p and no other objects, and if moreover this execution terminates, then the final state will satisfy the global assertion Q .

4 The proof system

The proof system we present consists of three levels. The first level, called the *local* proof system, enables one to reason about the correctness of an object. Testing the assumptions, which are introduced at the first level to deal with answer, send and new-statements, is done at the second level, which is called the *intermediate* proof system. The third level, the *global* proof system, formalizes the reasoning about a complete system.

4.1 The local proof system

The proof system for local correctness formulas dealing with assignment, sequential composition, the conditional and the loop construct equals the usual system for sequential programs:

Definition 4.1

We have the following well-known assignment axiom for instance variables:

$$(A, C : \{p^c[e_d^c/x_d^c]\}x_d^c := e_d^c\{p^c\}) \quad (\text{LIASS})$$

Definition 4.2

We have the following assignment axiom for temporary variables:

$$(A, C : \{p^c[e_d^c/u_d]\}u_d := e_d^c\{p^c\}) \quad (\text{LTASS})$$

The substitution operation occurring in the assignment axioms is the ordinary substitution, i.e., literal replacement of every occurrence of the variable x (u) by the expression e . Note that at this level we have no aliasing, i.e., there exist no two local expressions denoting the same variable.

Definition 4.3

The following rule formalizes reasoning about sequential composition:

$$\frac{(A, C : \{p^c\}S_1^c\{r^c\}), (A, C : \{r^c\}S_2^c\{q^c\})}{(A, C : \{p^c\}S_1^c; S_2^c\{q^c\})} \quad (\text{LSC})$$

Definition 4.4

Next we define the rule for the alternative command:

$$\frac{(A, C : \{p^c \wedge e\}S_1^c\{q^c\}), (A, C : \{p^c \wedge \neg e\}S_2^c\{q^c\})}{(A, C : \{p^c\}\text{if } e \text{ then } S_1^c \text{ else } S_2^c \text{ fi}\{q^c\})} \quad (\text{LALT})$$

Definition 4.5

We have the following rule for the iteration construct:

$$\frac{(A, C : \{p^c \wedge e\}S^c\{p^c\})}{(A, C : \{p^c\}\text{while } e \text{ do } S^c \text{ od}\{p^c \wedge \neg e\})} \quad (\text{LIT})$$

Definition 4.6

We have the following consequence rule:

$$\frac{p^c \rightarrow p_1^c, (A, C : \{p_1^c\}S^c\{q_1^c\}), q_1^c \rightarrow q^c}{(A, C : \{p^c\}S^c\{q^c\})} \quad (\text{LCR})$$

The following axioms and rule deal with bracketed sections.

Definition 4.7

We have the following axiom about bracketed sections containing new-statements:

$$(A, C : \{C(l_1)\}l_1; R_1; x \leftarrow \text{new}; R_2; l_2\{A(l_2)\}) \quad (\text{BN})$$

We have a similar axiom in case the identity of the newly created object is assigned to a temporary variable.

Definition 4.8

We have the following axiom about bracketed sections containing send-statements:

$$(A, C : \{C(l_1)\}l_1; R_1; l; x \leftarrow e_0!m(e_1, \dots, e_n); R_2; l_2\{A(l_2)\}) \quad (\text{BS})$$

where $x \notin IVar(C(l))$. We have a similar axiom in case the result of the send-expression is assigned to a temporary variable.

Definition 4.9

We have the following rule about bracketed sections containing answer-statements:

$$\frac{(\emptyset, \emptyset : \{p[\bar{y}/\bar{u}]\}S_i\{p[\bar{y}/\bar{u}]\}), i = 1, \dots, n}{(A, C : \{p \wedge C(l_1)\}l_1; \text{answer}(m_1, \dots, m_n); l_2\{p \wedge A(l_2)\})} \quad (\text{BA})$$

where \bar{u} is the sequence of the temporary variables occurring in p , and \bar{y} is a corresponding sequence of new instance variables, and $TVar(C(l_1), A(l_2)) = \emptyset$. Furthermore, S_i denotes the body of the method m_i .

The axioms (BN), (BS) and the rule (BA) extract from the set C the precondition and from the set A the postcondition using the labels which mark the beginning and the end of the bracketed section. The rule (BA) additionally incorporates the derivation of some invariance property of the answer-statement involved. Note that in the derivation of this property we are not allowed to use the sets of assumptions A and commitments C . To reason about the new, send, and answer statements in the derivation of some invariance property as required by the rule (BA) we introduce an invariance axiom:

Definition 4.10

We have the following invariance axiom:

$$(A, C : \{p\}S\{p\}) \quad (\text{INV})$$

where $Mod(S) \cap IVar(p) = \emptyset$.

In the cooperation test the applications of the axioms (BN), (BS) and the rule (BA) will be justified.

4.2 The intermediate proof system

In this subsection we present the proof system for the intermediate correctness formulas. This proof system is derived from the proof system for the language SPOOL, a sequential version of POOL [AB2].

4.2.1 The assignment axiom

We have the following assignment axiom:

Definition 4.11

Let $x \leftarrow e \in \text{Stat}^c$ and $z \in \text{LVar}_c$. We define

$$\{P[e[z/\text{self}]/z.x]\}(z, x \leftarrow e)\{P\} \quad (\text{IASS}).$$

First note that we have to transform the expression e to the global expression $e[z/\text{self}]$ and substitute this latter expression for $z.x$ because we consider the execution of the assignment $x \leftarrow e$ by the object denoted by z . Furthermore we have to define this substitution operation $[e[z/\text{self}]/z.x]$ because the usual one does not consider possible aliases of the expression $z.x$. For example, the expression $z'.x$, where z' differs syntactically from z , has to be substituted by $e[z/\text{self}]$ if the variables z and z' both refer to the same object, i.e., if $z \doteq z'$ holds.

Definition 4.12

Given a global expression g'_d , a logical variable z_c and a variable x , $x \in \text{LVar}_d^c$, we define for an arbitrary global expression g the substitution of the expression g' for $z_c.x$ in g by induction on the complexity of g . The result of this substitution we denote by $g[g'_d/z_c.x]$. Let $[\cdot]$ abbreviate $[g'_d/z_c.x]$:

$$\begin{aligned} z[\cdot] &= z \\ g[\cdot] &= g, \quad g = n, \text{nil}, \text{self}, \text{true}, \text{false} \\ (g.y)[\cdot] &= g[\cdot].y, \quad y \neq x \\ (g.x)[\cdot] &= \text{if } g[\cdot] = z_c \text{ then } g'_d \text{ else } g[\cdot].x \text{ fi} \\ &\vdots \end{aligned}$$

The omitted cases are defined directly from the application of the substitution to the subexpressions. This substitution operation is generalized to a global assertion in a straightforward manner, notation: $P[g'_d/z_c.x]$.

The most important aspect of this substitution is certainly the conditional expression that turns up when we are dealing with an expression of the form $g.x$. This is necessary because a certain form of aliasing, as described by the example above, is possible: After substitution it is possible that g refers to the object denoted by the logical variable z_c , so that $g.x$ is the same variable as $z_c.x$ and should be substituted by g' . It is also possible that, after substitution, g does not refer to the object denoted

by z_c , and in this case no substitution should take place. Since we can not decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides “dynamically”.

The intended meaning of this substitution operation is that the value of the substituted expression (assertion) in a state equals the value of the expression (assertion) in the state resulting from assigning the value of the expression g'_d to the variable x of the object denoted by z_c . A proof of the correctness of the substitution operation can be found in section 6.

4.2.2 The creation of new objects

We describe the meaning of a new statement by the following axiom:

Definition 4.13

Let $x \leftarrow \text{new} \in \text{Stat}^c$, the type of the variable x being $d \in C$. Furthermore let $z \in LVar_c$ and $z' \in LVar_d$ be two distinct variables. We define

$$\{P[z'/z.x][\text{new}/z']\}(z, x \leftarrow \text{new})\{P\} \quad (\text{NEW})$$

provided z' does not occur in P .

The calculation of the weakest precondition of an assertion with respect to the creation of a new object is done in two steps; the first of which consists of the substitution of a fresh variable z' for $z.x$. This substitution makes explicit all the possible aliases of the expression $z.x$. Next we carry out the substitution $[\text{new}/z']$. This substitution interprets the variable z' as the new object.

The definition of this latter substitution operation is complicated by the fact that the newly created object does not exist in the state just before its creation, so that in this state we can not refer to it. Assuming the new object to be referred to by the logical variable z' (in the state just after its creation) we however are able to carry out the substitution due to the fact that this variable z' can essentially occur only in a context where either one of its instances variables is referenced, or it is compared for equality with another expression. In both of these cases we can predict the outcome without having to refer to the new object.

Definition 4.14

Let $[\cdot]$ abbreviate $[\text{new}/z_c]$, we first define $g[\cdot]$ by induction on the complexity of g :

$$\begin{aligned}
 z[\cdot] &= z, z \neq z_c \\
 z_c[\cdot] &\text{ is undefined} \\
 g[\cdot] &= g \quad g = n, \text{nil}, \text{self}, \text{true}, \text{false} \\
 (z.x)[\cdot] &= z.x, z \neq z_c \\
 (z_c.x_d)[\cdot] &= \text{nil} \\
 (g.y.x)[\cdot] &= (g.y)[\cdot].x \\
 \left(\begin{array}{l} \text{if } g_0 \\ \text{then } g_1 \\ \text{else } g_2 \\ \text{fi} \end{array} \right) .x [\cdot] &= \begin{array}{l} \text{if } g_0[\cdot] \\ \text{then } (g_1.x)[\cdot] \\ \text{else } (g_2.x)[\cdot] \\ \text{fi} \end{array} \\
 (g : g')[\cdot] &= g[\cdot] : g'[\cdot] \\
 |g|[\cdot] &= |g[\cdot]| \\
 \left(\begin{array}{l} \text{if } g_0 \\ \text{then } g_1 \\ \text{else } g_2 \\ \text{fi} \end{array} \right) [\cdot] &= \begin{array}{l} \text{if } g_0[\cdot] \\ \text{then } g_1[\cdot] \\ \text{else } g_2[\cdot] \\ \text{fi} \end{array} \quad \begin{array}{l} \text{if the substitutions are defined,} \\ \text{undefined otherwise} \end{array} \\
 (g_1 \doteq g_2)[\cdot] &= g_1[\cdot] \doteq g_2[\cdot] \quad g_1, g_2 \neq z_c, \text{if} \dots \text{fi} \\
 (g_1 \doteq g_2)[\cdot] &= \text{false} \quad g_i = z_c, g_j \neq z_c, \text{if} \dots \text{fi}, i \neq j \in \{1, 2\} \\
 (g_1 \doteq g_2)[\cdot] &= \text{true} \quad g_1 = g_2 = z_c
 \end{aligned}$$

$$\begin{array}{lcl}
 \left(\begin{array}{l} \text{if } g_0 \\ \text{then } g_1 \\ \text{else } g_2 \\ \text{fi} \end{array} \doteq g_3 \right) [.] = & \begin{array}{l} \text{if } g_0[.] \doteq \text{nil} \\ \text{then } (g_3 \doteq \text{nil})[.] \\ \text{else if } g_0[.] \\ \quad \text{then } (g_1 \doteq g_3)[.] \\ \quad \text{else } (g_2 \doteq g_3)[.] \\ \quad \text{fi} \\ \text{fi} \end{array} & \\
 \\
 \left(\begin{array}{l} \text{if } g_0 \\ \text{then } g_1 \\ \text{else } g_2 \\ \text{fi} \end{array} \right) [.] = & \begin{array}{l} \text{if } g_0[.] \doteq \text{nil} \\ \text{then } (g_3 \doteq \text{nil})[.] \\ \text{else if } g_0[.] \\ \quad \text{then } (g_1 \doteq g_3)[.] \quad g_3 \neq \text{if} \dots \text{fi} \\ \quad \text{else } (g_2 \doteq g_3)[.] \\ \quad \text{fi} \\ \text{fi} \end{array} &
 \end{array}$$

We have the following proposition about this substitution operation applied to global expressions:

Proposition 4.15

For every global expression g , logical variable z_c , $g[\text{new}/z_c]$ is defined iff g is not of the form gz :

$$gz ::= z_c \mid \text{if } g_0 \text{ then } gz \text{ else } g_1 \text{ fi} \mid \text{if } g_0 \text{ then } g_1 \text{ else } gz \text{ fi}$$

In section 6 we prove that $g[\text{new}/z_c]$ equals the value of the expression g in the state resulting from the creation of a new object of class c , assuming this new object in this new state to be referred to by the variable z_c .

Next we define $P[\text{new}/z_c]$ by induction on the complexity of P .

Definition 4.16

Let, again, $[\cdot]$ abbreviate $[\text{new}/z_c]$.

$$\begin{aligned}
g_{\text{Bool}}[\cdot] & \quad \text{defined as above} \\
(\neg P)[\cdot] & = \neg(P[\cdot]) \\
(P_1 \wedge P_2)[\cdot] & = (P_1[\cdot] \wedge P_2[\cdot]) \\
(\forall z_a P)[\cdot] & = \forall z_a(P[\cdot]), \quad a \neq c, c^* \\
(\forall z'_c P)[\cdot] & = \forall z'_c(P[\cdot]) \wedge P[z_c/z'_c][\cdot], \quad z'_c \neq z_c \\
(\forall z_{c^*} P)[\cdot] & = \forall z_{c^*} \forall z_{\text{Bool}^*}(|z_{c^*}| \doteq |z_{\text{Bool}^*}| \rightarrow P[z_{\text{Bool}^*}, z_c/z_{c^*}][\cdot]) \\
(\exists z_a P)[\cdot] & = \exists z_a(P[\cdot]), \quad a \neq c, c^* \\
(\exists z'_c P)[\cdot] & = \exists z'_c(P[\cdot]) \vee P[z_c/z'_c][\cdot], \quad z'_c \neq z_c \\
(\exists z_{c^*} P)[\cdot] & = \exists z_{c^*} \exists z_{\text{Bool}^*}(|z_{c^*}| \doteq |z_{\text{Bool}^*}| \wedge P[z_{\text{Bool}^*}, z_c/z_{c^*}][\cdot])
\end{aligned}$$

Here we assume that z_{Bool^*} does not occur in P . The case of quantification over the type c of the newly created object can be explained as follows: Suppose we interpret the result of the substitution in a state in which the object denoted by z_c does not yet exist. In the first part of the substituted formula the bound variable z'_c thus ranges over all the old objects. In the second part the object to be created (the object denoted by z_c) is dealt with separately. This is done by first (literally) substituting the variable z_c for the quantified variable z'_c and then applying the substitution $[\text{new}/z_c]$. In this way the second part of the substituted formula expresses that the assertion P is valid in the new state (the state after the creation of the object denoted by z_c) when this variable z'_c is interpreted as the newly created object. Together the two parts of the substituted formula express quantification over the whole range of existing objects in the new state.

The idea of the substitution operation $[z_{\text{Bool}^*}, z_c/z_{c^*}]$ is that z_{Bool^*} and z_c together code a sequence of objects in the state just after the creation of the new object. At the places where z_{Bool^*} yields true the value of the coded sequence is the newly created object. Where z_{Bool^*} yields false the value of the coded sequence is the same as the value of z_{c^*} and where z_{Bool^*} delivers \perp the sequence also yields \perp .

Now $g[z_{\text{Bool}^*}, z_c/z_{c^*}]$ is defined as follows:

Definition 4.17

Let $[.]$ abbreviate $[z_{\text{Bool}^*}, z_c/z_{c^*}]$.

$$\begin{aligned}
 z_{c^*}[.] & \quad \text{isundefined} \\
 z[.] & = z, z \neq z_{c^*} \\
 g[.] & = g, g = n, \text{nil}, \text{self}, \text{tue}, \text{false} \\
 (g.x)[.] & = g[.].x \\
 & \quad \text{if } z_{\text{Bool}^*} : (g[.]) \\
 (z_{c^*} : g)[.] & = \text{then } z_c \\
 & \quad \text{else } z_{c^*} : (g[.]) \\
 & \quad \text{fi} \\
 (g_1 : g_2)[.] & = g_1[.] : g_2[.], g_1 \neq z_{c^*} \\
 (|z_{c^*}|)[.] & = |z_{c^*}| \\
 (|g|)[.] & = |g[.]|, g \neq z_{c^*} \\
 & \dots
 \end{aligned}$$

We have the following proposition:

Proposition 4.18

For an arbitrary global expression g the expression $g[z_{\text{Bool}^*}, z_c/z_{c^*}]$ is defined iff g is not of the form g' :

$$g' ::= z_{c^*} \mid \text{if } g_0 \text{ then } g' \text{ else } g_1 \text{ fi} \mid \text{if } g_0 \text{ then } g_1 \text{ else } g' \text{ fi}$$

In section 6 can be found a proof that the assertion $P[\text{new}/z_c]$ holds in a state iff P holds in the state resulting from the creation of a new object of class c , assuming the newly created object in this new state to be referred to by the variable z .

4.2.3 Some other rules

The rules for sequential composition, the alternative, the iterative construct, and the consequence rule are straightforward translations of the corresponding rules of the local proof system.

Definition 4.19

Let $S_1, S_2 \in \text{Stat}^c$ and $z \in \text{LVar}_c$.

$$\frac{\{P\}(z, S_1)\{R\}, \{R\}(z, S_2)\{Q\}}{\{P\}(z, S_1; S_2)\{Q\}} \quad (\text{ISC})$$

Definition 4.20

Let $\text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi} \in \text{Stat}^c$ and $z \in \text{LVar}_c$.

$$\frac{\{P \wedge e[z/\text{self}]\}(z, S_1)\{Q\}, \{P \wedge \neg e[z/\text{self}]\}(z, S_2)\{Q\}}{\{P\}(z, \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi})\{Q\}} \quad (\text{IALT})$$

Definition 4.21

Let $\text{while } e \text{ do } S \text{ od} \in \text{Stat}^c$ and $z \in \text{LVar}_c$.

$$\frac{\{P \wedge e[z/\text{self}]\}(z, S)\{P\}}{\{P\}(z, \text{while } e \text{ do } S \text{ od})\{P \wedge \neg e[z/\text{self}]\}} \quad (\text{IIT})$$

Definition 4.22

Let $S \in \text{Stat}^c$ and $z \in \text{LVar}_c$.

$$\frac{P \rightarrow P_1, \{P_1\}(z, S)\{Q_1\}, Q_1 \rightarrow Q}{\{P\}(z, S)\{Q\}} \quad (\text{ICR})$$

Finally, we have the following two rules describing the parallel execution of two objects:

Definition 4.23

$$\frac{\{P\}(z_1, S_1)\{R\}, \{R\}(z_2, S_2)\{Q\}}{\{P\}(z_1, S_1) \parallel (z_2, S_2)\{Q\}} \quad (\text{Par})$$

Definition 4.24

$$\frac{P \rightarrow P_1, \{P_1\}(z_1, S_1) \parallel (z_2, S_2)\{Q_1\}, Q_1 \rightarrow Q}{\{P\}(z_1, S_1) \parallel (z_2, S_2)\{Q\}} \quad (\text{Cpar})$$

4.3 The global proof system

In this section we describe the global proof system. We first define the notion of the cooperation test:

Definition 4.25

Let $\rho^c = \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} | c_n : S_n^{c_n} \rangle$ be bracketed (that is, every new, send, and answer statement of ρ^c occurs in a bracketed section), with $U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} = D_{1, \bar{m}_1}^{c_1} \dots D_{n-1, \bar{m}_{n-1}}^{c_{n-1}}$, where $D_{i, \bar{m}_i}^{c_i} = \langle m_{1, \bar{d}_1}^{i, c_i} \Leftarrow \mu_{1, \bar{d}_1}^{i, c_i}, \dots, m_{n_i, \bar{d}_{n_i}}^{i, c_i} \Leftarrow \mu_{n_i, \bar{d}_{n_i}}^{i, c_i} \rangle : S_i^{c_i}$. (We define $D_n = \langle \rangle : S_n^{c_n}$.) The specifications

$$(A_k, C_k : \{p_k^{c_k}\} S_k^{c_k} \{q_k^{c_k}\}), 1 \leq k \leq n$$

(with $TVar(p_k, q_k) = \emptyset$) *cooperate* with respect to some global invariant $I \in GAss$ iff

1. There are no occurrences in I of variables which occur at the left hand side of an assignment which is not contained in a bracketed section
2. $\vdash (A_k, C_k : \{p_k^{c_k}\} S_k^{c_k} \{q_k^{c_k}\}), 1 \leq k \leq n$
3. $\vdash (A_i, C_i : \{A_i(l_1)\} R\{C_i(l_2)\}), 1 \leq i \leq n$, where $m \Leftarrow R_1; l_1; R; l_2; R_2 \uparrow e \in D_i$.
4. Let $l_1; R; l_2$ be a bracketed section, occurring in D_i , containing the new-statement $x \Leftarrow \text{new}$. Furthermore, let $z \in LVar_{c_i}$ be a new variable. Then:

$$\vdash \{I \wedge p[\bar{y}/\bar{u}][z/\text{self}]\}(z, R[\bar{y}/\bar{u}])\{I \wedge q[\bar{y}/\bar{u}][z/\text{self}] \wedge p_j^{c_j}[z.x/\text{self}]\}$$

where \bar{u} is the sequence of the temporary variables occurring in $p = C_i(l_1)$, $q = A_i(l_2)$, and \bar{y} is a corresponding sequence of new instance variables. Furthermore, we assume the variable x to be of type c_j .

5. For $l_1; R_1; l; x \Leftarrow e_0!m(e_1, \dots, e_k); R_2; l_2$ occurring in D_i , and $l'_1; \text{answer}(\dots, m, \dots); l'_2$ occurring in D_j , such that the type of e_0 is c_j , with m declared as $R'_1; l'_1; R; l'_2; R'_2 \uparrow e$, we have

$$\begin{aligned} & \{I \wedge r_1[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r'_1[z_2/\text{self}] \wedge P\} \\ \vdash & (z_1, R_1[\bar{y}_1/\bar{u}_1]) \parallel (z_2, R'_1[\bar{y}_2/\bar{u}_2]) \\ & \{I \wedge r[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r''_1[\bar{y}_2/\bar{u}_2][z_2/\text{self}]\} \end{aligned}$$

and

$$\begin{aligned} & \{I \wedge r[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r''_2[\bar{y}_2/\bar{u}_2][z_2/\text{self}] \wedge Q\} \\ \vdash & (z_1, R_2[\bar{y}_1/\bar{u}_1]) \parallel (z_2, R'_2[\bar{y}_2/\bar{u}_2]) \\ & \{I \wedge r_2[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r'_2[z_2/\text{self}]\} \end{aligned}$$

where \bar{u}_1 is the sequence of the temporary variables occurring in r, r_1, r_2, R_1, R_2 , and \bar{y}_1 is the corresponding sequence of new instance variables, \bar{u}_2 is a sequence of the temporary variables occurring in $r'_1, r'_2, r''_1, r''_2, R'_1, R'_2$, and \bar{y}_2 is a corresponding sequence of new instance variables. The variables z_1 and z_2 are new variables, z_1 being of type c_i and z_2 being of type c_j . Furthermore, we have

- $C_i(l_1) = r_1, C_i(l) = r, A_i(l_2) = r_2$
- $C_j(l'_1) = r'_1, A_j(l'_2) = r'_2, A_j(l''_1) = r''_1, C_j(l''_2) = r''_2$

Finally, we have

- $P = e_0[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \doteq z_2 \wedge \bigwedge_i e_i[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \doteq z_2.y'_i \wedge \bigwedge_j z_2.y''_j \doteq \text{nil}$
- $Q = z_1 \doteq z_2.y'_1 \wedge z_1.x \doteq e[\bar{y}_2/\bar{u}_2][z_2/\text{self}]$

where $\bar{y}' \cup \bar{y}'' = \bar{y}_2$, with \bar{y}' being the instance variables corresponding to the formal parameters and \bar{y}'' corresponding to the local variables of m .

6. The following assertion holds:

$$p_n^{c_n}[z/\text{self}] \wedge \forall z'(z' \doteq z) \wedge \bigwedge_{1 \leq i < n} (\forall z_i \text{ false}) \rightarrow I$$

Here the variables z and z' are assumed to be of type c_n , the type of the root-object, the variable z_i is assumed to be of type c_i .

The syntactic restriction on occurrences of variables in the global invariant I implies the invariance of this assertion over those parts of the program which are not contained in a bracketed section. The clauses 4 and 5 imply among others the invariance of the global invariant over the bracketed sections.

This global invariant expresses some invariant properties of the dynamically evolving pointer structures arising during a computation of ρ . These properties are invariant in the sense that they hold whenever the program counter of every existing object is at a location outside a bracketed section. The above method to prove the invariance of the global invariant is based on the following semantical property of bracketed sections: Every computation of ρ can be rearranged such that at every time there is at most one object executing a bracketed section containing a new-statement, or a bracketed section belonging to an answer-statement.

Clause 2 verifies in an uniform manner the behaviour of the objects belonging to a class defined by the program.

Clause 3 verifies the behaviour of the methods, more precisely, the part of the body of a method excluding its prelude and postlude.

Clause 4 discharges assumptions about bracketed sections containing new-statements. Additionally the truth of the precondition of the local process of the new object is established. Note that by definition of a bracketed section we know that immediately after the execution of a bracketed section containing a new-statement $x := \text{new}$ the newly created object is referred to by x .

Clause 5 establishes the cooperation between two arbitrary matching bracketed sections, where two bracketed sections are said to match if they contain a send-statement and an answer-statement which match, i.e., the method name mentioned in the send-statement occurs in the answer-statement and the class of the object to which this method is sent equals that of the answer-statement.

The first correctness formula of clause 5 describes the activation of the rendezvous whilst the second one describes the termination of it.

The state before the rendezvous is characterized by

- the global invariant I , which describes the complete system,
- the precondition r'_1 of the answer statement, which describes the local state of the receiver, lifted to the global assertion language.
- the precondition r_1 of the bracketed section, containing the send statement, which describes the local state of the sender, also lifted to the global assertion language, and, finally,
- a global assertion P expressing that the sender indeed addresses the receiver and that the actual parameters are stored in the instance variables which denote at the level of the global assertion language the formal parameters.

Note that we have to introduce new instance variables which at the level of the global assertion language stand for the temporary variables of the sender and the receiver (remember that temporary variables do not exist in the global assertion language). The activation of the rendezvous then is described as the parallel execution of the prelude of the bracketed section containing the send-statement, and that of the method being executed. Note that the order in which these statements are executed does not matter because by definition the values of the expressions denoting the receiver and the actual parameters are not affected by the execution of the prelude of the bracketed section containing the send-statement. After the execution of these preludes the global invariant must hold and the local assertions lifted to the global assertion language, which are associated with the corresponding control points.

The state just before the termination of the rendezvous, which is described by the parallel execution of the postlude of the bracketed section containing the send-statement and that of the method being executed, is characterized by

- the global invariant,
- the local assertions r, r''_2 , which are associated with the corresponding control points, lifted to the level of the global assertion language,
- a global assertion Q expressing that this incarnation of the method has indeed been invoked by the object executing the bracketed section containing the send-statement (note that here we make use of the fact that the identity of the sender is sent as parameter) and, furthermore, that the result has been sent back.

Note that we may assume that the result has been sent back before the execution of the method has been completed because, by definition, the execution of the postlude of the method does not affected the result. Furthermore, the variable of the sender in which this result is to be stored, is not allowed to occur in the local assertion associated with the label marking the send-statement. After the execution of these postludes the global invariant must hold again together with the local assertions, which are associated with the corresponding control points, lifted to the global assertion language.

Clause 6 establishes the truth of the global invariant in the initial state. Note that the assertion $\forall z_c \text{false}$ expresses that there exist no objects of class c . The assertion $\forall z'(z' \doteq z)$ expresses that there exists precisely one object of class c_n .

One of the main points of the above definition is the formalization of reasoning about recursive answer statements. The basic pattern of reasoning about recursive procedure calls is given by the following rule ([Ba]):

$$\frac{\{p\}m\{q\} \vdash \{p\}S\{p\}}{\{p\}m\{q\}}$$

where m is a recursive procedure defined as S . Now let m be a method declared as $m \Leftarrow R_1; l_1; R; l_2; R_2 \uparrow e$, with the (labeled) answer statement $l; \text{answer}(m); l'$ occurring in R . Clause 3 then roughly amounts to proving that

$$\{p\}\text{answer}(m)\{q\} \vdash \{p_1\}R\{q_1\}$$

where p and q are the assertions associated with l and l' , and p_1, q_1 the assertions associated with l_1 and l_2 . To derive from this the conclusion $\{p\}\text{answer}(m)\{q\}$ we have to ensure that the execution of the prelude R_1 in a state satisfying p results in a state in which p_1 holds, and that the execution of the postlude R_2 in a state satisfying q_1 results in a state in which q holds. So essentially we have to prove $\{p\}R_1\{p_1\}$ and $\{q_1\}R_2\{q\}$. But the executions of R_1 and R_2 are dependent upon the actual parameters sent. This dependency is taken care of by clause 5, where all the different possibilities for the actual parameters are accounted for by considering

the matching send statements. Summerizing, one can say that the clauses 3 and 5 together embody the following recursive rule for answer statements:

$$\frac{\{p\}\text{answer}(m)\{q\} \vdash \{p_1\}R\{q_1\}}{\{p\}\text{answer}(m)\{q\}}$$

where m, R, p, q, p_1 , and q_1 are defined as above, assuming for the sake of simplicity that in R there occur no new and send statements and no other other answer statements.

Finally for $\rho^c = \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} | c_n : S_n^{c_n} \rangle$, with $U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} = D_{1\bar{m}_1}^{c_1}, \dots, D_{n-1\bar{m}_{n-1}}^{c_{n-1}}$, where $D_{i\bar{m}_i}^{c_i} = \langle m_{1\bar{d}_1}^{i c_i} \Leftarrow \mu_{1\bar{d}_1}^{i c_i}, \dots, m_{n_i\bar{d}_{n_i}}^{i c_i} \Leftarrow \mu_{n_i\bar{d}_{n_i}}^{i c_i} \rangle : S_i^{c_i}$ we have the following rules:

Definition 4.26

We have the following program rule:

$$\frac{(A_i, C_i : \{p_i^{c_i}\})S_i^{c_i}\{q_i^{c_i}\}, 1 \leq i \leq n, \text{ cooperate w.r.t. } I}{\{p_n^{c_n}[z/\text{self}]\}\rho\{I \wedge \bigwedge_{1 \leq i < n} \forall z_i q_i^{c_i}[z_i/\text{self}] \wedge q_n^{c_n}[z/\text{self}]\}} \quad (\text{PR})$$

where z is of type c_n and z_i is of type c_i .

Note that in the conclusion of the program rule (PR) we take as precondition the precondition of the local process of the root-object because initially only this object exists. The postcondition consists of a conjunction of the global invariant, the assertions $\forall z_i q_i^{c_i}[z_i/\text{self}]$, which express that the final local state of every object of class c_i is characterized by the local assertion $q_i^{c_i}$, and the assertion $q_n^{c_n}[z/\text{self}]$ expressing that the final local state of the root-object is characterized by the local assertion $q_n^{c_n}$.

Definition 4.27

We have the following consequence rule for programs:

$$\frac{p^{c_n} \rightarrow p_1^{c_n}, \{p_1^{c_n}[z_{c_n}/\text{self}]\}\rho\{Q_1\}, Q_1 \rightarrow Q}{\{p^{c_n}[z_{c_n}/\text{self}]\}\rho\{Q\}} \quad (\text{PC})$$

Definition 4.28

Next we have a substitution rule to initialize instance variables:

$$\frac{\{p^{c_n}[z_{c_n}/\text{self}]\}\rho\{Q\}}{\{(p^{c_n}[l/x])[z_{c_n}/\text{self}]\}\rho\{Q\}} \quad (\text{S1})$$

provided the instance variable x does not occur in ρ or Q .

Definition 4.29

The following substitution rule initializes logical variables:

$$\frac{\{p^{c_n}[z_{c_n}/\text{self}]\}\rho\{Q\}}{\{(p^{c_n}[l/z])[z_{c_n}/\text{self}]\}\rho\{Q\}} \quad (\text{S2})$$

provided the logical variable z does not occur in Q .

Definition 4.30

The following rule is used to describe the initial state:

$$\frac{\{(p^{c_n} \wedge x \doteq \text{nil})[z_{c_n}/\text{self}]\}\rho\{Q\}}{\{p^{c_n}[z_{c_n}/\text{self}]\}\rho\{Q\}} \quad (\text{INIT})$$

where $x \in \bigcup_c IVar_c^{c_n}$.

Definition 4.31

Finally, we have the following rule for *auxiliary variables*:

$$\frac{\{P\}\rho'\{Q\}}{\{P\}\rho\{Q\}} \quad (\text{AUX})$$

where ρ is obtained from ρ' by deleting all assignments to variables belonging to some set Aux , i.e. a set of auxiliary variables, such that for an arbitrary assignment $x \leftarrow e$ ($u \leftarrow e$) occurring in ρ' we have that $ITvar(e) \cap Aux \neq \emptyset$ implies that $x \in Aux$ ($u \in Aux$), moreover, the variables of the set Aux do not occur in tests of ρ' or in assignments $x \leftarrow s$ ($u \leftarrow s$), s not a simple expression, and, finally, $IVar(Q) \cap Aux = \emptyset$.

The rule for auxiliary variables can be explained as follows: To be able to express some properties of a program ρ it may be necessary to add some assignments to new variables, which are called auxiliary variables. These assignments may not influence the flow of control of ρ , otherwise these auxiliary variables can not be used to express some properties of ρ . This requirement is formulated syntactically.

5 Semantics

In this section we define in a formal way the semantics of the programming language and the assertion languages. First, in section 5.1, we deal with the assertion languages on their own. Then, in section 5.2, we give a formal semantics to the programming language, making use of *transition systems*. Finally, section 5.3 formally defines the notion of truth of a correctness formula.

5.1 Semantics of the assertion languages

For every type $a \in C^\dagger$, we shall let \mathbf{O}^a denote the set of objects of type a , with typical element α^a . To be precise, we define $\mathbf{O}^{\text{Int}} = \mathbf{Z}$ and $\mathbf{O}^{\text{Bool}} = \mathbf{B}$, whereas for every class $c \in C$ we just take for \mathbf{O}^c an arbitrary infinite set. With \mathbf{O}_\perp^d we shall denote $\mathbf{O}^d \cup \{\perp\}$, where \perp is a special element not in \mathbf{O}^d , which will stand for ‘undefined’, among others the value of the expression `nil`. Now for every type $d \in C^+$ we let \mathbf{O}^{d^*} denote the set of all finite sequences of elements from \mathbf{O}_\perp^d and we take $\mathbf{O}_\perp^{d^*} = \mathbf{O}^{d^*}$. This means that sequences can contain \perp as a component, but a sequence can never be \perp itself (as an expression of a sequence type, `nil` just stands for the empty sequence).

Definition 5.1

We shall often use generalized Cartesian products of the form

$$\prod_{i \in A} B(i).$$

As usual, the elements of this set are the functions f with domain A such that $f(i) \in B(i)$ for every $i \in A$.

Definition 5.2

Given a function $f \in A \rightarrow B$, $a \in A$, and $b \in B$, we use the *variant notation* $f\{b/a\}$ to denote the function in $A \rightarrow B$ that satisfies

$$f\{b/a\}(a') = \begin{cases} b & \text{if } a' = a \\ f(a') & \text{otherwise.} \end{cases}$$

Definition 5.3

The set *GState* of *global states*, with typical element σ , is defined as follows:

$$GState = \left(\prod_d P^d \right) \times \prod_c (\mathbf{O}^c \rightarrow \prod_d (IVar_d^c \rightarrow \mathbf{O}_\perp^d)) \times \prod_c (\mathbf{O}^c \rightarrow (\prod_d (TVar_d \rightarrow \mathbf{O}_\perp^d))^*)$$

where P^c , for every $c \in C$, denotes the set of finite subsets of \mathbf{O}^c , and for $d = \text{Int}, \text{Bool}$ we define $P^d = \mathbf{O}^d$.

A global state describes the situation of a complete system of objects at a certain moment during program execution. The first component specifies for each class the set of *existing* objects of that class, that is, the set of objects that have been created up to this point in the execution of the program. Relative to some global state σ an object $\alpha \in \mathbf{O}^d$ can be said to exist if $\alpha \in \sigma_{(1)(d)}$. For the built-in data types we have for every global state σ that $\sigma_{(1)}(\text{Int}) = \mathbf{Z}$ and $\sigma_{(1)}(\text{Bool}) = \mathbf{B}$. Note that $\perp \notin \sigma_{(1)(d)}$ for every $d \in C^+$. The second component of a global state specifies for each object the values of its instance variables. The third component specifies for each object a *stack* of local environments, i.e., functions assigning objects to temporary variables.

We introduce the following abbreviations:

- We abbreviate $\sigma_{(1)(d)}$ to $\sigma^{(d)}$.
- The local state of an object α in σ we will denote by $\sigma(\alpha)$, it consists of the assignment of objects to the instance variables and the temporary variables as given by $\sigma_{(2)}(\alpha)$, $Top(\sigma_{(3)}(\alpha))$, respectively. Furthermore, $\sigma(\alpha)(x)$ and $\sigma(\alpha)(u)$ will abbreviate $\sigma_{(2)(c,d)}(\alpha)(x)$, $Top(\sigma_{(3)(c)}(\alpha))(d)(u)$, respectively, assuming the type of α to be c and that of x and u to be d . Here $Top(< f_1, \dots, f_n >) = f_n$.
- Furthermore, $\sigma\{\beta/\alpha, x\}$ will denote the state resulting from σ by assigning β to the variable x of α , and $\sigma\{\beta/\alpha, u\}$ will denote the state resulting from assigning β to the variable u of the top local environment of α , i.e., $Top(\sigma_{(3)}(\alpha))$.

Definition 5.4

The set $LState^c$ of *local states* of class c , with typical element θ , is defined by

$$LState^c = \mathbf{O}^c \times GState$$

For convenience sake we formalize the notion of a local state as a pair consisting of an object name and a global state, instead of an object name and a function characterizing the values of the variables of that object.

Definition 5.5

We now define the set $LEnv$ of *logical environments*, with typical element ω , by

$$LEnv = \prod_a (LVar_a \rightarrow \mathbf{O}_\perp^a).$$

A logical environment assigns values to logical variables. We abbreviate $\omega_{(a)}(z_a)$ to $\omega(z_a)$.

Definition 5.6

The following semantic functions are defined in a straightforward manner. We omit most of the detail and only give the most important cases:

1. The function $\mathcal{E}_d^c \in Exp_d^c \rightarrow LState^c \rightarrow \mathbf{O}_\perp^d$ assigns a value $\mathcal{E}[\![e]\!](\theta)$ to the expression e_d^c in the local state θ^c . For example, $\mathcal{E}_d^c[\![nil]\!](\theta) = \perp$ and $\mathcal{E}_d^c[\![x_d^c]\!](\langle \alpha, \sigma \rangle) = \sigma(\alpha)(x_d^c)$.
2. The function $\mathcal{L}_d^c \in LExp_d^c \rightarrow LEnv \rightarrow LState^c \rightarrow \mathbf{O}_\perp^d$ assigns a value $\mathcal{L}[\![l]\!](\omega)(\theta)$ to the local expression l_d^c in the logical environment ω and the local state θ^c .
3. The function $\mathcal{G}_a \in GExp_a \rightarrow LEnv \rightarrow GState \rightarrow \mathbf{O}_\perp^a$ assigns a value $\mathcal{G}[\![g]\!](\omega)(\sigma)$ to the global expression g_a in the logical environment ω and the global state σ .

4. The function $\mathcal{A}^c \in LAss^c \rightarrow LEnv \rightarrow LState^c \rightarrow \mathbf{B}$ assigns a value $\mathcal{A}[p](\omega)(\theta)$ to the local assertion p^c in the logical environment ω and the local state θ^c . Here the following cases are special:

$$\mathcal{A}[l_{\text{Bool}}](\omega)(\theta) = \begin{cases} \mathbf{t} & \text{if } \mathcal{L}[l](\omega)(\theta) = \mathbf{t} \\ \mathbf{f} & \text{if } \mathcal{L}[l](\omega)(\theta) = \mathbf{f} \text{ or } \mathcal{L}[l](\omega)(\theta) = \perp \end{cases}$$

$$\mathcal{A}[\exists z_d p](\omega)(\theta) = \begin{cases} \mathbf{t} & \text{if there is an } \alpha^d \in \mathbf{O}^d \text{ such that } \mathcal{A}[p](\omega\{\alpha/z\})(\theta) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases}$$

Note that in the latter case $d = \text{Int}$ or $d = \text{Bool}$ and that the range of quantification *does not include* \perp .

5. The function $\mathcal{A} \in GAss \rightarrow LEnv \rightarrow GState \rightarrow \mathbf{B}$ assigns a value $\mathcal{A}[P](\omega)(\sigma)$ to the global assertion P in the logical environment ω and the global state σ . The following cases are special:

$$\mathcal{A}[g_{\text{Bool}}](\omega)(\sigma) = \begin{cases} \mathbf{t} & \text{if } \mathcal{L}[g](\omega)(\sigma) = \mathbf{t} \\ \mathbf{f} & \text{if } \mathcal{L}[g](\omega)(\sigma) = \mathbf{f} \text{ or } \mathcal{L}[g](\omega)(\sigma) = \perp \end{cases}$$

$$\mathcal{A}[\exists z_d P](\omega)(\sigma) = \begin{cases} \mathbf{t} & \text{if there is an } \alpha^d \in \sigma^{(d)} \text{ such that } \mathcal{A}[P](\omega\{\alpha/z\})(\sigma) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases}$$

Note that here d can be any type in C^+ and that the quantification ranges over $\sigma^{(d)}$, the set of *existing* objects of type d (which does not include \perp).

$$\mathcal{A}[\exists z_{d^*} P](\omega)(\sigma) = \begin{cases} \mathbf{t} & \text{if there is an } \alpha^{d^*} \in \mathbf{O}^{d^*} \text{ such} \\ & \text{that } \alpha(n) \in \sigma^{(d)} \cup \perp \text{ for all } n \in \mathbf{N} \\ & \text{and } \mathcal{A}[P](\omega\{\alpha/z\})(\sigma) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases}$$

For sequence types, quantification ranges over those sequences of which every element is either \perp or an existing object.

The values $\mathcal{G}[g_a](\omega)(\sigma)$ of the global expression g_a and $\mathcal{A}[g](\omega)(\sigma)$ of the global assertion P are in fact only meaningful for those ω and σ that are consistent and compatible:

Definition 5.7

We define the global state σ to be *consistent*, for which we use the notation $OK(\sigma)$ iff

$$\forall c \in C \forall \alpha \in \sigma^{(c)} \forall d \in C \forall x \in IVar_d^c \sigma(\alpha)(x) \in \sigma^{(d)} \cup \perp.$$

and

$$\forall c \in C \forall \alpha \in \sigma^{(c)} \forall d \in C \forall u \in TVar_d \sigma(\alpha)(u) \in \sigma^{(d)} \cup \perp$$

In other words, the value in σ of a variable of an existing object is either \perp or an existing object itself.

Furthermore we define the logical environment ω to be *compatible* with the global state σ , with the notation $OK(\omega, \sigma)$, iff $OK(\sigma)$ and, additionally,

$$\forall d \in C \forall z \in LVar_d (\omega(z) \in \sigma^{(d)} \cup \{\perp\})$$

and

$$\forall d \in C \forall z \in LVar_d \forall a \in \mathbb{N} (\omega(z)(n) \in \sigma^{(d)} \cup \{\perp\}).$$

In other words, ω assigns to every logical variable z_d of a simple type the value \perp or an existing object, and to every sequence variable z_{d^*} a sequence of which each element is an existing object or equals \perp .

5.2 The transition system

We will describe the internal behaviour of an object by means of a transition system. A *local configuration* we define to be a pair (S^c, θ^c) . The set of local configurations is denoted by $LConf$. Let $Rec = \{ \langle \alpha, \beta \rangle, \langle m, \bar{\beta} \rangle, \langle m^O, \bar{\beta} \rangle, \langle m^I, \bar{\beta} \rangle : \alpha, \beta \in \bigcup_c O^c, \bar{\beta} \text{ denoting a sequence of objects} \} \cup \{\epsilon\}$. A pair $\langle \alpha, \beta \rangle$ is called an *activation record*. It records the information that the object α created β . Sequences of the form $\langle m, \bar{\beta} \rangle, \langle m^I, \bar{\beta} \rangle$, and $\langle m^O, \bar{\beta} \rangle$ are called *communication records*. A record $\langle m, \beta_0, \dots, \beta_n \rangle$, with n the number of formal parameters of m , records the information that the method m has been sent by β_1 to β_0 with actual parameters β_1, \dots, β_n . (Remember that the identity of the sender is sent as the first actual parameter.) On the other hand a record $\langle m, \beta_1, \dots, \beta_n \rangle$, with again n the number of formal parameters, records the information that the method m has been received with actual parameters β_1, \dots, β_n . A record $\langle m^O, \beta_1, \beta_2 \rangle$ records the information that the result of the method m , the object β_2 , has been sent to β_1 . A record $\langle m^I, \beta_1, \beta_2 \rangle$ records the information that the result of the method m , the object β_2 , has been received from β_1 .

We define for every $r \in Rec$ a transition relation $\rightarrow^r \subseteq LConf \times LConf$. (In fact we define \rightarrow^r given a unit U .) To facilitate the semantics we introduce the auxiliary statement E , the empty statement, to denote termination, the statement $send(m, e, \alpha)$ and the expression $wait(m, \beta)$. The statement $send(m, e, \alpha)$ will model the process of sending the result of m , the value of e , to α . The expression $wait(m, \beta)$ will model the process of waiting for the object β to send the result of m . Furthermore, we introduce the operations *Push* and *Pop*:

$$\begin{aligned} Push(\langle f_1, \dots, f_n \rangle, f) &= \langle f_1, \dots, f_n, f \rangle \\ Pop(\langle f_1, \dots, f_n \rangle) &= \langle f_1, \dots, f_{n-1} \rangle. \end{aligned}$$

Definition 5.8

Let $\theta = \langle \alpha, \sigma \rangle$. We define

- $(x_d \leftarrow e_d, \theta) \rightarrow^\epsilon (E, \theta')$,
where $\theta' = \langle \alpha, \sigma\{\mathcal{E}\llbracket e_d \rrbracket(\theta)/\alpha, x\}\rangle$.
- $(u_d \leftarrow e_d, \theta) \rightarrow^\epsilon (E, \theta')$,
where $\theta' = \langle \alpha, \sigma\{\mathcal{E}\llbracket e_d \rrbracket(\theta)/\alpha, u\}\rangle$.
- $(x_d := \text{new}, \theta) \rightarrow^{<\alpha, \beta>} (E, \theta')$,
where $\theta' = \langle \alpha, \sigma'\{\beta/\alpha, x_d\}\{\perp/\beta, y\}_{y \in \text{IVar}^d}\rangle$, and

$$\sigma' = (\sigma_{(1)}\{\sigma^{(d)} \cup \{\beta\}\}, \sigma_{(2)}, \sigma_{(3)}),$$

and $\beta \in \mathbf{O}^d \setminus \sigma^{(d)}$.

- $(u_d := \text{new}, \theta) \rightarrow^{<\alpha, \beta>} (E, \theta')$,
where $\theta' = \langle \alpha, \sigma'\{\beta/\alpha, u_d\}\{\perp/\beta, y\}_{y \in \text{IVar}^d}\rangle$, and

$$\sigma' = (\sigma_{(1)}\{\sigma^{(d)} \cup \{\beta\}\}, \sigma_{(2)}, \sigma_{(3)}),$$

and $\beta \in \mathbf{O}^d \setminus \sigma^{(d)}$.

- $(x \leftarrow e_0!m(e_1, \dots, e_n), \theta) \rightarrow^{<m, \bar{\beta}>} (x \leftarrow \text{wait}(m, \beta_0), \theta)$,
where $\beta_i = \mathcal{E}\llbracket e_i \rrbracket(\theta)$, and $\bar{\beta} = \beta_0, \dots, \beta_n$.
- $(u \leftarrow e_0!m(e_1, \dots, e_n), \theta) \rightarrow^{<m, \bar{\beta}>} (u \leftarrow \text{wait}(m, \beta_0), \theta)$,
where $\beta_i = \mathcal{E}\llbracket e_i \rrbracket(\theta)$, and $\bar{\beta} = \beta_0, \dots, \beta_n$.
- $(x \leftarrow \text{wait}(m, \beta_0), \theta) \rightarrow^{<m^I, \beta_0, \gamma>} (E, \theta')$,
where $\theta' = \langle \alpha, \sigma\{\gamma/\alpha, x_d\}\rangle$, with γ an arbitrary element of \mathbf{O}_\perp^d .
- $(u \leftarrow \text{wait}(m, \beta_0), \theta) \rightarrow^{<m^I, \beta_0, \gamma>} (E, \theta')$,
where $\theta' = \langle \alpha, \sigma\{\gamma/\alpha, x_d\}\rangle$.
- $(\text{answer}(\dots, m, \dots), \theta) \rightarrow^{<m, \bar{\beta}>} (S; \text{send}(m, e, \beta_1), \theta')$,
where $\theta' = \langle \alpha, \sigma\{\text{Push}(\sigma_{(3)}(\alpha), f)/\alpha\}\rangle$, and

$$\begin{aligned} f(u_i) &= \beta_i & \text{if } u_i \in \{u_1, \dots, u_n\} \\ f(u) &= \perp & \text{if } u \notin \{u_1, \dots, u_n\}. \end{aligned}$$

Furthermore,

$$\sigma\{\text{Push}(\sigma_{(3)}(\alpha), f)/\alpha\} = (\sigma_{(1)}, \sigma_{(2)}, \sigma_{(3)}\{\text{Push}(\sigma_{(3)}(\alpha), f)/\alpha\}).$$

Here we assume u_1, \dots, u_n to be the formal parameters of m , The statement S to be its body, and e to be its result expression.

- $(\text{send}(m, e, \beta), \theta) \rightarrow^{<m^O, \beta, \gamma>} (E, \theta')$,
where $\theta' = \langle \alpha, \sigma\{\text{Pop}(\sigma_{(3)}(\alpha)/\alpha)\} \rangle$, and $\gamma = \mathcal{E}[\![e]\!](\theta)$. Here

$$\sigma\{\text{Pop}(\sigma_{(3)}(\alpha)/\alpha)\} = (\sigma_{(1)}, \sigma_{(2)}, \sigma_{(3)}\{\text{Pop}(\sigma_{(3)}(\alpha)/\alpha)\}).$$

- $(!; S, \theta) \rightarrow^\epsilon (S, \theta)$.
- $\frac{(S_1, \theta) \rightarrow^r (S_2, \theta') \mid (E, \vartheta')}{(S_1; S, \theta) \rightarrow^r (S_2; S, \theta') \mid (S, \vartheta')}$
- $(\text{if } e_{\text{Bool}} \text{ then } S_1 \text{ else } S_2 \text{ fi}, \theta) \rightarrow^\epsilon (S_1, \theta)$,
if $\mathcal{E}[\![e_{\text{Bool}}]\!](\theta) = \text{true}$.
- $(\text{if } e_{\text{Bool}} \text{ then } S_1 \text{ else } S_2 \text{ fi}, \theta) \rightarrow^\epsilon (S_2, \theta)$,
if $\mathcal{E}[\![e_{\text{Bool}}]\!](\theta) = \text{false}$.
- $(\text{while } e_{\text{Bool}} \text{ do } S \text{ od}, \theta) \rightarrow^\epsilon (S; \text{while } e_{\text{Bool}} \text{ do } S \text{ od}, \theta)$,
if $\mathcal{E}[\![e_{\text{Bool}}]\!](\theta) = \text{true}$.
- $(\text{while } e_{\text{Bool}} \text{ do } S \text{ od}, \theta) \rightarrow^\epsilon (E, \theta)$,
if $\mathcal{E}[\![e_{\text{Bool}}]\!](\theta) = \text{false}$.

We define $\rightarrow^h = TC(\bigcup_{r \in \text{Rec}} \rightarrow^r)$. Here the operation TC denotes the transitive and reflexive closure which composes additionally the communication records and activation records into a *history* h , a sequence of communication records and activation records.

Using the above transition system we define another transition relation \rightarrow_L which *hides* the computations within the bracketed sections.

Definition 5.9

- $\frac{(S_1, \theta_1) \rightarrow^r (S_2, \theta_2)}{(S_1, \theta_1) \rightarrow_L (S_2, \theta_2)}$,
where S_1 is not of the form $S'; S''$, with S' a bracketed section.
- $\frac{(S, \theta_1) \rightarrow^h (E, \theta_2)}{(!; S; !'; S', \theta_1) \rightarrow_L (!'; S', \theta_2)}$,
where $!; S; !'$ is a bracketed section.

The semantics of statements and local correctness formulas will be defined with respect to this transition relation \rightarrow_L .

Next we describe the behaviour of several objects working in parallel. The local behaviour of the objects we shall derive from the local transition system as described above. But at this level we have the necessary information to select the right choices concerning the communications.

We define an *intermediate configuration* to be a tuple $(\sigma, (\alpha_i, S_i^{\alpha_i})_i)$, where $\alpha_i \in \sigma^{(c_i)}$, assuming all the α_i to be distinct. The set of intermediate configurations will be denoted by $IConf$. We define $\rightarrow^r \subseteq IConf \times IConf$, with $r = \langle m, \bar{\beta} \rangle$, as follows (note that we use the same notation as for the local transition relation, however this will cause no harm):

Definition 5.10

We define

- $$\frac{(S_j, \langle \alpha_j, \sigma \rangle) \rightarrow^r (S'_j, \langle \alpha_j, \sigma' \rangle)}{(\sigma, (\alpha_i, S_i)_i) \rightarrow^r (\sigma', (\alpha_i, S'_i)_i)}$$

where $r = \epsilon, \langle \alpha_j, \beta \rangle$

$$S'_i = S_i \quad i \neq j$$

$$= S'_i \quad \text{otherwise.}$$
- $$\frac{(S_j, \langle \alpha_j, \sigma \rangle) \rightarrow^r (S'_j, \langle \alpha_j, \sigma_1 \rangle), (S_k, \langle \alpha_k, \sigma \rangle) \rightarrow^{r'} (S'_k, \langle \alpha_k, \sigma_2 \rangle)}{(\sigma, (\alpha_i, S_i)_i) \rightarrow^{r''} (\sigma', (\alpha_i, S'_i)_i)}$$

where $j \neq k$ and $r = \langle m, \beta_1, \dots, \beta_n \rangle$, $r' = \langle m, \alpha_j, \beta_1, \dots, \beta_n \rangle$, furthermore, we have

$$S'_i = S_i \quad i \neq j, k$$

$$= S'_i \quad i = j, k,$$

and $\sigma' = \sigma\{\sigma_1(\alpha_j)/\alpha_j\}\{\sigma_2(\alpha_k)/\alpha_k\}$. (Here $\sigma\{\sigma_1(\alpha_j)/\alpha_j\}\{\sigma_2(\alpha_k)/\alpha_k\}$ denotes the state resulting from changing the local state of α_j (α_k) to $\sigma_1(\alpha_j)$ ($\sigma_2(\alpha_k)$).)
- $$\frac{(S_j, \langle \alpha_j, \sigma \rangle) \rightarrow^r (S'_j, \langle \alpha_j, \sigma_1 \rangle), (S_k, \langle \alpha_k, \sigma \rangle) \rightarrow^{r'} (S'_k, \langle \alpha_k, \sigma_2 \rangle)}{(\sigma, (\alpha_i, S_i)_i) \rightarrow^{r''} (\sigma', (\alpha_i, S'_i)_i)}$$

where $j \neq k$ and $r = \langle m^O, \alpha_k, \gamma \rangle$, $r' = \langle m^I, \alpha_j, \gamma \rangle$ and $r'' = \langle m, \alpha_j, \gamma \rangle$, furthermore, we have

$$S'_i = S_i \quad i \neq j, k$$

$$= S'_i \quad i = j, k,$$

and $\sigma' = \sigma\{\sigma_1(\alpha_j)/\alpha_j\}\{\sigma_2(\alpha_k)/\alpha_k\}$.

The first rule above selects one object and its local state and uses the local transition system to derive one local step of this object. The second and third rule select two objects which are ready to communicate with each other. Note that we use a different interpretation of a communication record now: A record $\langle m, \alpha, \beta_1, \dots, \beta_n \rangle$, with n the number of formal parameters of m , records the information that α has received

the method m with actual parameters β_1, \dots, β_n , where β_1 in fact denotes the sender. Furthermore, $\langle m, \alpha, \gamma \rangle$ will now record the information that α has received the result of m , the object γ .

We define $\rightarrow^h = TC(\bigcup_{r \in Rec} \rightarrow^r)$, again using the same notation as for the transitive and reflexive closure of the local transition relation, from the context however it should be clear which one is meant. This new transition relation will be used to define the semantics of intermediate correctness formulas.

To describe the behaviour of a complete system we introduce the notion of a *global configuration*: a pair (X, σ) , where $X \in \prod_c \mathbf{O}^c \rightarrow Stat^c$, and a transition relation $\rightarrow^r \subseteq GConf \times GConf$. We note again that we do not notationally distinguish between the different transition relations, from the context however it will be clear which one is meant. The set of all global configurations we denote by $GConf$. We will abbreviate in the sequel $X_{(c)}(\alpha)$, for $\alpha \in \mathbf{O}^c$, by $X(\alpha)$. The idea is that $X(\alpha)$ denotes the statement to be executed by α .

Definition 5.11

We have the following rule

$$\frac{(\sigma, (\alpha_i, X(\alpha_i))_i) \rightarrow^r (\sigma', (\alpha_i, S_i^{c_i})_i)}{(X, \sigma) \rightarrow^r (X', \sigma')}$$

where $\alpha_i \in \sigma^{(c_i)}$ (all the α_i distinct) and $X' = X\{S_i^{c_i}/\alpha_i\}_i$.

This rule selects some finite set of objects which execute in parallel according to the previous transition system.

We define $\rightarrow^h = TC(\bigcup_{r \in Rec} \rightarrow^r)$. This transition relation will be used to define the semantics of programs and global correctness formulas.

We proceed with the following definition which characterizes the set of *initial* and *final* global configurations of a given program ρ :

Definition 5.12

Let $\rho^c = \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} | c_n : S_n^{c_n} \rangle$, with $U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} = D_{1\bar{m}_1}^{c_1}, \dots, D_{n-1\bar{m}_{n-1}}^{c_{n-1}}$, where $D_{i\bar{m}_i}^{c_i} = \langle m_{1\bar{d}_1}^{i c_i} \Leftarrow \mu_{1\bar{d}_1}^{i c_i}, \dots, m_{n_i\bar{d}_{n_i}}^{i c_i} \Leftarrow \mu_{n_i\bar{d}_{n_i}}^{i c_i} \rangle : S_i^{c_i}$. Furthermore let $X \in \prod_c \mathbf{O}^c \rightarrow Stat^c$. We define

$$Init_\rho(X) \text{ iff}$$

- $X(\alpha) = S_i^{c_i}$, $c_i \in \{c_1, \dots, c_n\}$, $\alpha \in \mathbf{O}^{c_i}$.
- $X(\alpha) = E$, $\alpha \in \mathbf{O}^c$, $c \notin \{c_1, \dots, c_n\}$.

We define for a state σ such that $OK(\sigma)$:

$Init_\rho(\sigma)$ iff

- $\sigma^{(c)} = \emptyset \quad c \in \{c_1, \dots, c_{n-1}\}$
 $\quad = \{\alpha\} \quad c = c_n, \text{ for some } \alpha \in \mathbf{O}^{c_n}$
- $\sigma(\alpha)(x) = \perp$, for $\alpha \in \sigma^{(c_n)}$ and $x \in IVar_{c_n}^{c_n}$.

We next define $Init_\rho((X, \sigma))$ iff $Init_\rho(X)$ and $Init_\rho(\sigma)$. We define

$Final_\rho((X, \sigma))$ iff $X(\alpha) = E$, for all $\alpha \in \sigma^{(c_i)}$, $c_i \in \{c_1, \dots, c_n\}$

The predicate $Init_\rho(Final_\rho)$ characterizes the set of *initial (final)* configurations of ρ . Note that the value of a variable $x_c^{c_n}$ of the root-object, $c \in \{c_1, \dots, c_n\}$, is undefined initially. This follows for $c \neq c_n$ from the fact that we consider only consistent states and that initially only the root-object exists (with respect to the classes c_1, \dots, c_n). But the consistency of the initial state would also allow the value of a variable $x \in IVar_{c_n}^{c_n}$ to be the root-object itself. However, as it will appear to be convenient with respect to the formulation of some rules which formalize reasoning about the initial state, we define the initial state to be completely specified by the variables ranging over the standard objects.

Now we are able to define the meaning of the following programming constructs: S^c , (z_c, S^c) , $(z_{c_i}, S_1^{c_i}) \parallel (z_{c_j}, S_2^{c_j})$ and ρ .

Definition 5.13

We define

$$S[S^c](\theta) = \{ \langle (S_1, \theta_1), \dots, (S_n, \theta_n) \rangle : (S_i, \theta_i) \rightarrow_L (S_{i+1}, \theta_{i+1}), 1 \leq i < n, S_1 = S, \theta_1 = \theta \}$$

Definition 5.14

We define $\mathcal{I}[(z, S)](\omega)(\sigma_1) = \emptyset$ if not $OK(\omega, \sigma_1)$, and $\mathcal{I}[(z_1, S_1) \parallel (z_2, S_2)](\omega)(\sigma_1) = \emptyset$ if not $OK(\omega, \sigma_1)$ or $\omega(z_1) = \omega(z_2)$. So assume from now on that $OK(\omega, \sigma_1)$, furthermore that $\omega(z) = \alpha$, $\omega(z_i) = \alpha_i$.

$$\mathcal{I}[(z, S)](\omega)(\sigma_1) = \{ \sigma_2 : \text{for some } h \ (\sigma_1, (\alpha, S)) \rightarrow^h (\sigma_2, (\alpha, E)) \}$$

Assuming furthermore that $\alpha_1 \neq \alpha_2$:

$$\begin{aligned} \mathcal{I}[(z_1, S_1) \parallel (z_2, S_2)](\omega)(\sigma_1) = \\ \{\sigma_2 : \text{for some } h \ (\sigma_1, (\alpha_1, S_1), (\alpha_2, S_2)) \rightarrow^h (\sigma_2, (\alpha_1, E), (\alpha_2, E))\} \end{aligned}$$

Definition 5.15

The semantics of programs is defined as follows:

$$\mathcal{P}[\rho](\sigma_1) = \{\sigma_2 : \text{for some } h \ (X_1, \sigma_1) \rightarrow^h (X_2, \sigma_2)\}$$

where $Init_\rho((X_1, \sigma_1))$ and $Final_\rho((X_2, \sigma_2))$.

Note that $\mathcal{P}[\rho](\sigma) = \emptyset$ if it is not the case that $Init_\rho(\sigma)$.

5.3 Truth of correctness formulas

In this section we define formally the truth of the local, intermediate, and global correctness formulas, respectively. First we define the truth of local correctness formulas.

Definition 5.16

We define

$$\models (A, C : \{p^c\} S^c \{q^c\}) \text{ iff}$$

for every ω and $\langle (S_1, \theta_1), \dots, (S_n, \theta_n) \rangle \in S[S](\theta_1)$:

- $\theta_1, \omega \models p$
- $\theta_i, \omega \models A(Lab(S_i)), 1 \leq i \leq n$

implies

$$\theta_n, \omega \models C(Lab(S_n)) \text{ and if } S_n = E \text{ then } \theta_n, \omega \models q.$$

Here

$$\begin{aligned} Lab(S) &= \mathbf{l} \quad \text{if } S = \mathbf{l}; S' \\ &= \emptyset \quad \text{otherwise} \end{aligned}$$

(Note that for X a set of labeled assertions we have $X(\emptyset) = \text{true}$.)

Next we define the truth of intermediate correctness formulas.

Definition 5.17

We define

$$\models \{P\}(z_c, S^c)\{Q\} \text{ iff} \\ \forall \omega \forall \sigma_1 \forall \sigma_2 \in \mathcal{I}[(z_c, S^c)](\omega)(\sigma_1) : \sigma_1, \omega \models P \Rightarrow \sigma_2, \omega \models Q.$$

And

$$\models \{P\}(z_{c_i}, S^{c_i}) \parallel (z'_{c_j}, S^{c_j})\{Q\} \text{ iff} \\ \forall \omega \forall \sigma_1 \forall \sigma_2 \in \mathcal{I}[(z_{c_i}, S^{c_i}) \parallel (z'_{c_j}, S^{c_j})](\omega)(\sigma_1) : \sigma_1, \omega \models P \Rightarrow \sigma_2, \omega \models Q.$$

Finally, we define the truth of global correctness formulas.

Definition 5.18

We define

$$\models \{P\}\rho\{Q\} \text{ iff } \forall \omega \forall \sigma_1 \forall \sigma_2 \in \mathcal{P}[\rho](\sigma_1) : \sigma_1, \omega \models P \Rightarrow \sigma_2, \omega \models Q.$$

6 Soundness

In this section we prove the soundness of the proof system as presented in the previous section. We first discuss the soundness of the local proof system.

6.1 The local proof system

The following theorem states the every local correctness formula derivable from the the local proof system is valid:

Theorem 6.1

For every local correctness formula $(A, C : \{p\}S\{q\})$ we have
if

$$\vdash (A, C : \{p\}S\{q\})$$

then

$$\models (A, C : \{p\}S\{q\})$$

(Here \vdash denotes derivability from the local proof system.)

The above theorem is proved by induction on the length of the derivation, that is, we prove the soundness of the axioms, and for every rule we prove that the validity of the premisses implies the validity of the conclusion. We treat the rule BA, the other axioms and rules being straightforward to deal with.

Lemma 6.2

If

$$\models (\emptyset, \emptyset : \{p[\bar{y}/\bar{u}]\} S_i : \{p[\bar{y}/\bar{u}]\})$$

then

$$\models (A, C : \{p \wedge C(l_1)\} l_1; \text{answer}(m_1, \dots, m_n); l_2 \{p \wedge A(l_2)\}).$$

Proof

Let $R = l_1; \text{answer}(m_1, \dots, m_n); l_2$. As the semantic function S is defined with respect to the transition relation \rightarrow_L , which hides the computations within bracketed sections, it is sufficient to consider for every θ the case of $\langle (R, \theta), (l_2, \theta') \rangle \in S[R](\theta)$ such that

- $\theta, \omega \models p \wedge C(l_1)$,
- $\theta', \omega \models A(l_2)$.

We then have to prove that $\theta', \omega \models p$. Let $(\text{answer}(m_1, \dots, m_n), \theta_1) \rightarrow^{r_1} \dots \rightarrow^{r_{k-1}} (E, \theta_k)$, with $\theta_1 = \theta$ and $\theta_k = \theta'$. We next define for $1 \leq i \leq k$, $\theta'_i = \langle \alpha, \sigma'_i \rangle$ (assuming $\theta_i = \langle \alpha, \sigma_i \rangle$), with $\sigma'_i = \sigma_i \{ \sigma_1(\alpha)(\bar{u}) / \alpha, \bar{y} \}$. It then follows that: $(\text{answer}(m_1, \dots, m_n), \theta'_1) \rightarrow^{r_1} \dots \rightarrow^{r_{k-1}} (E, \theta'_k)$ (note that \bar{y} are some new variables not occurring in the body of m_i , $1 \leq i \leq n$). We have for some $1 \leq i \leq n$ that $(S_i, \theta'_2) \rightarrow^{r_2} \dots \rightarrow^{r_{k-2}} (E, \theta'_{k-1})$ and $\theta'_2, \omega \models p[\bar{y}/\bar{u}]$, with S_i the body of method m_i : σ'_2 results from σ'_1 by creating a new local environment for the execution of S_i by α , and σ'_k is obtained from σ'_{k-1} by popping the stack of local environments associated with α . (Note that σ'_2 and σ'_1 agree with respect to the instance variables of α). From this and $\models (\emptyset, \emptyset : \{p[\bar{y}/\bar{u}]\} S_i : \{p[\bar{y}/\bar{u}]\})$ we infer that $\theta'_{k-1}, \omega \models p[\bar{y}/\bar{u}]$. Since $p[\bar{y}/\bar{u}]$ contains no temporary variables and σ'_k and σ'_{k-1} agree with respect to the instance variables of α we have $\theta'_k, \omega \models p[\bar{y}/\bar{u}]$, or, equivalently, $\theta_k, \omega \models p$ (note that $\sigma'_k(\alpha)(\bar{y}) = \sigma'_1(\alpha)(\bar{y}) = \sigma_1(\alpha)(\bar{u}) = \sigma'_1(\alpha)(\bar{u}) = \sigma'_k(\alpha)(\bar{u})$). We conclude $\theta', \omega \models p$. \square

6.2 The intermediate proof system

The following theorem states that every intermediate correctness formula derivable from the intermediate proof system is valid:

Theorem 6.3

Whenever

$$\vdash \{P\}(z, R)\{Q\}$$

then

$$\models \{P\}(z, R)\{Q\}$$

and, whenever

$$\vdash \{P\}(z, R) \parallel (z', R')\{Q\}$$

then

$$\models \{P\}(z, R) \parallel (z', R')\{Q\}$$

We prove the validity of the assignment axiom (IASS) and the axiom (NEW). The proof of the above theorem then proceeds by a straightforward induction on the length of the derivation. To prove the soundness of the assignment axiom (IASS) we need the following lemma about the correctness of the corresponding substitution operation. This lemma states that semantically substituting the expression g' for $z.x$ in an assertion (expression) yields the same result when evaluating the assertion (expression) in the state where the value of g' is assigned to the variable x of the object denoted by z .

Lemma 6.4

For an arbitrary σ, ω such that $OK(\omega, \sigma)$ we have:

$$\mathcal{G}\llbracket g[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) = \mathcal{G}\llbracket g \rrbracket(\omega)(\sigma')$$

and

$$\mathcal{A}\llbracket P[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) = \mathcal{A}\llbracket P \rrbracket(\omega)(\sigma')$$

where $\sigma' = \sigma\{\mathcal{G}\llbracket g'_d \rrbracket(\omega)(\sigma)/\omega(z_c), x_d\}$.

Proof

By induction on the complexity of g and P . We treat only the case $g = g_1.x$, all the other ones following directly from the induction hypothesis. Now:

$$\begin{aligned} \mathcal{G}\llbracket g[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) &= \\ \mathcal{G}\llbracket \text{if } g_1[g'_d/z_c.x_d] \doteq z_c \text{ then } g' \text{ else } g_1[g'/z_c.x].x \text{ fi} \rrbracket(\omega)(\sigma) \end{aligned}$$

Suppose that $\mathcal{G}\llbracket g_1[g'_d/z_c.x_d] \rrbracket(\omega)(\sigma) = \omega(z_c)$. We have: $\mathcal{G}\llbracket g_1.x \rrbracket(\omega)(\sigma') = \sigma'(\mathcal{G}\llbracket g_1 \rrbracket(\omega)(\sigma'))(x)$. So by the induction hypothesis we have that:

$$\mathcal{G}\llbracket g_1.x \rrbracket(\omega)(\sigma') = \sigma'(\omega(z_c))(x) = \mathcal{G}\llbracket g'_d \rrbracket(\omega)(\sigma).$$

On the other hand if $\mathcal{G}[g_1[g'_d/z_c.x_d]](\omega)(\sigma) \neq \omega(z_c)$ then:

$$\begin{aligned}
\mathcal{G}[g_1[g'_d/z_c.x_d].x_d](\omega)(\sigma) &= \\
\sigma(\mathcal{G}[g_1[g'_d/z_c.x_d]](\omega)(\sigma))(x_d) &= \text{(definition of } \sigma') \\
\sigma'(\mathcal{G}[g_1[g'_d/z_c.x_d]](\omega)(\sigma))(x_d) &= \text{(induction hypothesis)} \\
\sigma'(\mathcal{G}[g_1](\omega)(\sigma'))(x_d) &= \\
\mathcal{G}[g_1.x_d](\omega)(\sigma').
\end{aligned}$$

□

The following lemma states the soundness of the axiom (IASS)

Lemma 6.5

We have

$$\models \{P[e[z/\text{self}]/z.x]\}(z, x \leftarrow e)\{P\},$$

where we assume $x \leftarrow e \in \text{Stat}^c$ and $z \in \text{LVar}_c$.

Proof

Let σ, ω , with $OK(\omega, \sigma)$, such that $\sigma, \omega \models P[e[z/\text{self}]/z.x]$ and $\sigma' \in \mathcal{I}[(z, x \leftarrow e)](\omega)(\sigma)$. It follows that $\sigma' = \sigma\{\mathcal{G}[e[z/\text{self}]](\omega)(\sigma)/\omega(z), x\}$ (note that as e is a local expression we have $\mathcal{G}[e[z/\text{self}]](\omega)(\sigma) = \mathcal{E}[e](\langle \alpha, \sigma(\alpha) \rangle)$, with $\alpha = \omega(z)$, a formal proof of which proceeds by a straightforward induction on the complexity of e). Thus by the previous lemma we conclude $\sigma', \omega \models P$. □

To prove the soundness of the axiom describing the new statement we need the following lemma which states the correctness of the corresponding substitution operation. This lemma states that semantically the substitution $[\text{new}/z]$ applied to an assertion yields the same result when evaluating the assertion in the state resulting from the creation of a new object, interpreting the variable z as the newly created object.

Lemma 6.6

For an arbitrary $\omega, \omega', \sigma, \sigma', \beta \in \mathbf{O}^c \setminus \sigma^{(c)}$ such that $OK(\omega, \sigma)$ and

$$\sigma' = (\sigma_{(1)}\{\sigma^{(c)} \cup \{\beta\}/c\}, \sigma_{(2)}\{\perp/\beta, y\}_{y \in \text{IVar}^c}, \sigma_{(3)})$$

and $\omega' = \omega\{\beta/z_c\}$, we have for an arbitrary assertion P :

$$\mathcal{A}[P[\text{new}/z_c]](\omega)(\sigma) = \mathcal{A}[P](\omega')(\sigma').$$

The proof of this lemma proceeds by induction on the structure of P . To carry out this induction argument, which we trust the interested reader to be able to perform,

we need the following two lemmas. The first of which is applied to the case $P = g$ and the second of which is applied to the case $P = \exists zP'$, $z \in LVar_a$, $a = c, c^*$.

Lemma 6.7

For an arbitrary σ, ω , with $OK(\omega, \sigma)$, global expression g and logical variable z_c such that $g[new/z_c]$ is defined we have:

$$\mathcal{G}\|g\|(\omega')(\sigma') = \mathcal{G}\|g[new/z_c]\|(\omega)(\sigma)$$

where $\sigma' = (\sigma_{(1)}\{\sigma^{(c)} \cup \{\beta\}/c\}, \sigma_{(2)}\{\perp/\beta, y\}_{y \in IVar^c}, \sigma_{(3)})$ and $\omega' = \omega\{\beta/z_c\}$, $\beta \notin \sigma^{(c)}$.

Proof

Induction on the structure of g . □

The following lemma states that semantically the substitution $[z_{Bool^*}, z_c/z_{c^*}]$ applied to an assertion (expression) yields the same result when updating the sequence denoted by the variable z_{c^*} to the value of z_c at those positions for which the sequence denoted by z_{Bool^*} gives the value true.

Lemma 6.8

Let $\omega, \sigma, \alpha = \omega(z_{c^*})$, $\alpha' = \omega(z_{Bool^*})$ such that $|\alpha| = |\alpha'|$ and $OK(\omega, \sigma)$.

Let $\alpha'' \in \mathbf{O}^{c^*}$ such that

- $|\alpha''| = |\alpha|$
- for $n \in \mathbf{N}$: $\alpha''(n) = \omega(z_c)$ if $\alpha'(n) = \text{true}$
 $= \alpha(n)$ if $\alpha'(n) = \text{false}$
 $= \perp$ if $\alpha'(n) = \perp$

Let $\omega' = \omega\{\alpha''/z_{c^*}\}$. Then:

1. For every g such that $g[z_{Bool^*}, z_c/z_{c^*}]$ is defined:

$$\mathcal{G}\|g[z_{Bool^*}, z_c/z_{c^*}]\|(\omega)(\sigma) = \mathcal{G}\|g\|(\omega')(\sigma)$$

2. For every P such that z_{Bool^*} does not occur in it:

$$\mathcal{A}\|P[z_{Bool^*}, z_c/z_{c^*}]\|(\omega)(\sigma) = \mathcal{A}\|P\|(\omega')(\sigma)$$

Proof

Induction on the structure of g and P . □

Now we are ready to prove the soundness of the axiom (NEW).

Lemma 6.9

We have

$$\models \{P[z'/z.x][\text{new}/z']\}(z, x := \text{new})\{P\},$$

where $x := \text{new} \in \text{Stat}^c$, $z \in \text{LVar}_c$, and z' is a new logical variable of the same type as x .

Proof

Let σ, ω , with $OK(\sigma, \omega)$, such that $\sigma, \omega \models P[z'/z.x][\text{new}/z']$ and $\sigma' \in \mathcal{I}[(z, x := \text{new})](\omega)(\sigma)$. We have by lemma 6.6 that $\sigma'', \omega' \models P[z'/z.x]$, where $\omega' = \omega\{\beta/z'\}$, with $\beta \in \mathbf{O}^d \setminus \sigma^d$, assuming d to be the type of the variable x , and $\sigma'' = (\sigma_{(1)}\{\sigma^{(d)} \cup \{\beta\}/d\}, \sigma_{(2)}\{\perp/\beta, y\}_{y \in \text{LVar}^c}, \sigma_{(3)})$. Now by lemma 6.4 it follows that $\sigma', \omega' \models P$. Finally, as z' does not occur in P we have $\sigma', \omega \models P$. □

6.3 The global proof system

The following theorem states that every global correctness formula derivable from the global proof system is valid:

Theorem 6.10

Whenever

$$\vdash \{p[z/\text{self}]\}\rho\{Q\}$$

then

$$\models \{p[z/\text{self}]\}\rho\{Q\}$$

We will prove only the validity of the rule (PR), the other rules being straightforward to deal with. We first introduce some definitions.

Definition 6.11

We call a global configuration (X, σ) *stable* iff there exists no object executing inside a bracketed section. With respect to the global configuration (X, σ) an object $\alpha \in \sigma^{(c)}$, for some c , is said to be executing inside a bracketed section if $X(\alpha) = R; R'$, for some R' , with R a substatement of a prelude or postlude of a bracketed section containing a new or send statement, or the prelude, postlude of the body of a method. Note that if $X(\alpha) = \text{wait}(m, \beta); R'$, for some method m and object β , that is, α has just finished executing the prelude of a bracketed section containing a send statement and

has sent the actual parameters to β , then α is *not* considered to be executing within a bracketed section.

Definition 6.12

We call a global computation of ρ , i.e., a sequence $(X_1, \sigma_1), \dots, (X_n, \sigma_n)$ such that $Init_\rho((X_1, \sigma_1))$, and for $1 \leq i < n$ we have $(X_i, \sigma_i) \rightarrow^{r_i} (X_{i+1}, \sigma_{i+1})$, for some record r_i , *regular* if in every configuration (X_i, σ_i) at most one object is executing inside a bracketed section.

We observe that every terminating computation of a program ρ (ρ arbitrary) can be rearranged into an equivalent (with respect to the local behaviour of the objects) regular one. More precisely, for every terminating computation $(X_1, \sigma_1), \dots, (X_n, \sigma_n)$ of a program ρ there exists a regular computation $(X'_1, \sigma'_1), \dots, (X'_k, \sigma'_k)$ such that for every object α the sequence $(X_1(\alpha), \sigma_1(\alpha)), \dots, (X_n(\alpha), \sigma_n(\alpha))$ equals the sequence $(X'_1(\alpha), \sigma'_1(\alpha)), \dots, (X'_k(\alpha), \sigma'_k(\alpha))$ modulo *finite stuttering*.

It is not difficult to see that the following lemma implies the soundness of the rule (PR):

Lemma 6.13

Let $\rho^c = \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} | c_n : S_n^{c_n} \rangle$ be bracketed, with $U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} = D_{1, m_1}^{c_1}, \dots, D_{n-1, m_{n-1}}^{c_{n-1}}$, where $D_{i, m_i}^{c_i} = \langle m_{1, d_1}^{i, c_i} \Leftarrow \mu_{1, d_1}^{i, c_i}, \dots, m_{n_i, d_{n_i}}^{i, c_i} \Leftarrow \mu_{n_i, d_{n_i}}^{i, c_i} \rangle : S_i^{c_i}$. Let $(A_i, C_i : \{p_i^{c_i}\} S_i^{c_i} \{q_i^{c_i}\})$, $1 \leq i \leq n$, be some cooperating (with respect to some global invariant I) specifications. Then for an arbitrary regular computation $(X_0, \sigma_0), \dots, (X_k, \sigma_k)$ of ρ such that $\langle \alpha, \sigma_0 \rangle, \omega \models p_n$ (α being the root-object), and (X_k, σ_k) is stable, we have

- for every object $\alpha \in \sigma_k^{(c_i)}$ we have $\langle \alpha, \sigma_k \rangle, \omega \models A_i(Lab(X_k(\alpha)))$,
- $\sigma_k, \omega \models I$,
- if $\alpha \in \mathbf{O}^{c_i}$ is a newly created object of σ_k then $\langle \alpha, \sigma_k \rangle, \omega \models p_i^{c_i}$.

Proof

The proof proceeds by induction on the length of the history h associated with the computation $(X_0, \sigma_0), \dots, (X_k, \sigma_k)$. Let us consider first the case that $|h| = 0$. Let α be the root object. As the history h is empty and (X_k, σ_k) is stable we have that $X_k(\alpha) \notin A_n$ (note that $l \in A_n$ marks the rear of a bracketed section), so we have that $\langle \alpha, \sigma_k \rangle, \omega \models A_n(Lab(X_n(\alpha)))$ holds vacuously. Furthermore, as the variables of I can only be changed by the execution of a bracketed section (by the first clause of the cooperation test), we have $\sigma_k, \omega \models I$ (note that $\sigma_0, \omega \models I$ by $\langle \alpha, \omega \rangle, \omega \models p_n$ and the last clause of the cooperation test).

Now let $|h| > 0$. First we consider the following case: Let

$$(X_0, \sigma_0) \rightarrow \dots \rightarrow (X_m, \sigma_m) \rightarrow \dots \rightarrow (X_k, \sigma_k)$$

such that

- $X_m(\alpha) = l_1; R_1; l; x \leftarrow e_0!m(\bar{e}); R_2; l_2; S$, for some S , with $\alpha \in \mathbf{O}^{c_i}$
- $X_m(\beta) = l'_1; \text{answer}(\dots, m, \dots); l'_2; S'$, for some S' , with $\beta \in \mathbf{O}^{c_j}$
- From (X_m, σ_m) to (X_k, σ_k) only α and β are executing; α is executing the statement R_1 and sending β the actual parameters, and β is executing the statement R'_1 , assuming m to be declared as $R'_1; l''_1; R; l''_2; R'_2 \uparrow e$.

Let

$$\begin{aligned} r_1 &= C_i(l_1), & r'_1 &= C_j(l'_1), \\ r &= C_i(l), & r''_1 &= A(l''_1). \end{aligned}$$

Next suppose that $l_1; R_1; l; x \leftarrow e_0!m(\bar{e}); R_2; l_2$ occurs in the statement $S_i^{c_i}$. Let k' be such that from $(X_{k'}, \sigma_{k'})$ to (X_m, σ_m) α is executing $S_i^{c_i}$. From the induction hypothesis it then follows that $\langle \alpha, \sigma_{k'} \rangle, \omega \models p_i^{c_i}$. We are given that $\models (A_i, C_i : \{p_i^{c_i}\} S_i^{c_i} \{q_i^{c_i}\})$ (by the second clause of the cooperation test and the soundness of the local proof system), so from the truth definition of local correctness formulas and another application of the induction hypothesis we have that $\langle \alpha, \sigma_m \rangle, \omega \models r_1$. On the other hand if $l_1; R_1; l; x \leftarrow e_0!m(\bar{e}); R_2; l_2$ occurs in the body of some method m' it follows in a similar way from $\models (A_i, C_i : \{A(l_1)\} \bar{R}\{C_i(l_2)\})$, assuming m' to be declared as $\bar{R}_1; \bar{l}_1; \bar{R}; \bar{l}_2; \bar{R}_2 \uparrow e'$, and the induction hypothesis, that $\langle \alpha, \sigma_m \rangle, \omega \models r_1$.

Analogously we have that $\langle \beta, \sigma_m \rangle, \omega \models r'_1$.

Furthermore, as X_m is stable we have by the induction hypothesis that $\sigma_m, \omega \models I$.

Next we define

$$\sigma'_m = \sigma_m \{ \sigma_m(\alpha)(\bar{u}_1)/\alpha, \bar{y}_1 \} \{ \mathcal{E}[e_i](\langle \alpha, \sigma_m \rangle) / \beta, y'_i \}_{1 \leq i \leq n} \{ \perp / \beta, y''_j \}_j,$$

where $\bar{y}_2 = \bar{y}' \cup \bar{y}''$, with \bar{y}' being a sequence of instance variables corresponding to the formal parameters of m and \bar{y}'' to the local variables of m . It then follows that

$$\sigma'_m, \omega' \models I \wedge r_1[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r'_1[z_2/\text{self}] \wedge P,$$

where $\omega' = \omega \{ \alpha, \beta / z_1, z_2 \}$ and $P = e_0[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \doteq z_2 \wedge \bigwedge_i e_i[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \doteq z_2.y'_i \wedge \bigwedge_j z_2.y''_j \doteq \text{nil}$. Next, let

$$(\sigma'_m, (\alpha, R_1[\bar{y}_1/\bar{u}_1]), (\beta, R'_1[\bar{y}_2/\bar{u}_2])) \rightarrow^* (\sigma', (\alpha, E), (\beta, E)).$$

It then follows by the cooperation test and the soundness of the intermediate proof system that

$$\sigma', \omega \models I \wedge \bar{r}[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r''_1[\bar{y}_2/\bar{u}_2][z_2/\text{self}].$$

Now it is not difficult to see that

$$\sigma_k = \sigma' \{ \sigma_m(\alpha)(\bar{y}_1)/\alpha, \bar{y}_1 \} \{ \sigma_m(\beta)(\bar{y}_2)/\beta, \bar{y}_2 \} \{ \sigma'(\alpha)(\bar{y}_1)/\alpha, \bar{u}_1 \} \{ Push(\sigma'_{(2)(\beta)}, f)/\beta \},$$

where $f(\bar{u}_2) = \sigma'(\beta)(\bar{y}_2)$. So we conclude that

- $\sigma_k, \omega \models I$
- $\langle \alpha, \sigma_k \rangle, \omega \models r$
- $\langle \beta, \sigma_k \rangle, \omega \models r''_1$.

Next we consider the following case. Let

$$(X_0, \sigma_0) \rightarrow \dots (X_m, \sigma_m) \rightarrow \dots \rightarrow (X_k, \sigma_k)$$

such that

- $X_m(\alpha) = x \leftarrow \text{wait}(m, \beta); R_2; l_2; S$, for some S , with $\alpha \in \mathbf{O}^{c_i}$
- $X_m(\beta) = l'_2; R'_2; \text{send}(m, e, \alpha); S'$, for some S' , assuming R'_2 to be the postlude of the method m , with m declared as $R'_1; l'_1; R; l'_2; R'_2 \uparrow e$, and $\beta \in \mathbf{O}^{c_j}$
- From (X_m, σ_m) to (X_k, σ_k) only α and β are executing; α is executing $x \leftarrow \text{wait}(m, \beta); R_2$ and β is executing $R'_2; \text{send}(m, e, \alpha)$.

Let $C_j(l'_2) = r''_2$ and $A_j(l'_2) = r'_2$, where l'_2 marks the end of the answer statement which gave rise to the activation of the method m . Furthermore, let $A_i(l_2) = r_2$. From $\models (A_j, C_j : \{r'_1\}R\{r''_2\})$ and the induction hypothesis we infer that $\langle \beta, \sigma_m \rangle, \omega \models r''_2$. Moreover, by the induction hypothesis we have $\sigma_m, \omega \models I$. From the previous case it follows that we may assume $\langle \alpha, \sigma_m \rangle, \omega \models r$, where $r = C_i(l)$, with l the label marking the send statement which activated the method m . Next, we define

$$\sigma'_m = \sigma_m \{ \sigma_m(\alpha)(\bar{u}_1)/\alpha, \bar{y}_1 \} \{ \mathcal{E}[e](\langle \beta, \sigma_m \rangle)/\alpha, x \} \{ \sigma_m(\beta)(\bar{u}_2)/\beta, \bar{y}_2 \}.$$

It then follows that

$$\sigma'_m, \omega' \models I \wedge r[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r''_2[\bar{y}_2/\bar{u}_2][z_2/\text{self}] \wedge Q,$$

where $\omega' = \omega \{ \alpha, \beta/z_1, z_2 \}$, and $Q = z_1 \doteq z_2.y'_1 \wedge z_1.x \doteq e[\bar{y}_2/\bar{u}_2][z_2/\text{self}]$. (Note that x does not occur in r .) Next, let

$$(\sigma'_m, (\alpha, R_2[\bar{y}_1/\bar{u}_1]), (\beta, R'_2[\bar{y}_2/\bar{u}_2])) \rightarrow^* (\sigma', (\alpha, E), (\beta, E)).$$

It then follows by the cooperation test and the soundness of intermediate proof system that

$$\sigma', \omega' \models I \wedge r_2[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge r'_2[z_2/\text{self}].$$

It is not difficult to see that

$$\sigma_k = \sigma' \{ \sigma_m(\alpha)(\bar{y}_1)/\alpha, \bar{y}_1 \} \{ \sigma_m(\beta)(\bar{y}_2)/\beta, \bar{y}_2 \} \{ \sigma'(\alpha)(\bar{y}_1)/\alpha, \bar{u}_1 \} \{ Pop(\sigma'_{(2)(\beta)})/\beta \}.$$

So we conclude that

- $\sigma_k, \omega \models I$
- $\langle \alpha, \sigma_k \rangle, \omega \models r_2$
- $\langle \beta, \sigma_k \rangle, \omega \models r'_2.$

(Note that $TVar(r'_2) = \emptyset$.)

Finally, we have the following case to consider: Let

$$(X_0, \sigma_0) \rightarrow \dots \rightarrow (X_m, \sigma_m) \rightarrow \dots \rightarrow (X_k, \sigma_k)$$

such that

- $X_m(\alpha) = l_1; R_1; x \leftarrow \text{new}; R_2; l_2; S$, for some S , with $\alpha \in \mathbf{O}^{c_i}$.
- From (X_m, σ_m) to (X_k, σ_k) α is executing $l_1; R_1; x \leftarrow \text{new}; R_2; l_2$.

Let $C_i(l_1) = r_1$ and $A_i(l_2) = r_2$. As in the previous cases, by the induction hypothesis we have

$$\langle \alpha, \sigma_m \rangle, \omega \models r_1 \text{ and } \sigma_m, \omega \models I.$$

Next, we define

$$\sigma'_m = \sigma_m \{ \sigma_m(\alpha)(\bar{u})/\alpha, \bar{y} \}.$$

It then follows that

$$\sigma'_m, \omega \models I \wedge r_1[\bar{y}/\bar{u}][z/\text{self}].$$

where $\omega' = \omega \{ \alpha/z \}$. Now, let

$$(\sigma'_m, (\alpha, (R_1; x \leftarrow \text{new}; R_2)[\bar{y}/\bar{u}])) \rightarrow^* (\sigma', (\alpha, E)).$$

It then follows by the fourth clause of the cooperation test and the soundness of the intermediate proof system that

$$\sigma', \omega' \models I \wedge r_2[z, \bar{y}/\text{self}, \bar{u}] \wedge p[z \cdot x/\text{self}],$$

where p is the precondition of the newly created object. It is not difficult to see that

$$\sigma_k = \sigma' \{ \sigma_m(\alpha)(\bar{y})/\alpha, \bar{y} \} \{ \sigma'(\alpha)(\bar{y})/\alpha, \bar{u} \}.$$

So we conclude that $\langle \alpha, \sigma_k \rangle, \omega \models r_2$, $\langle \beta, \sigma_k \rangle, \omega \models p$ (here β is the newly created object), and $\sigma_k, \omega \models I$. \square

7 Completeness

In this section we prove that every valid correctness formula about a program is derivable. To be more specific, let $\{p[z/\text{self}]\rho\{Q\}$ be a valid correctness formula, with $\rho^c = \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} | c_n : S_n^c \rangle$ where $U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} = D_{1, m_1}^{c_1}, \dots, D_{n-1, m_{n-1}}^{c_{n-1}}$, and $D_{i, m_i}^{c_i} = \langle m_{1, d_1}^{i, c_i} \Leftarrow \mu_{1, d_1}^{i, c_i}, \dots, m_{n_i, d_{n_i}}^{i, c_i} \Leftarrow \mu_{n_i, d_{n_i}}^{i, c_i} \rangle : S_i^{c_i}$. (Here we put $D_n = \langle \rangle : S_n^c$.) Without loss of generality we assume that $IVar(p, Q) \subseteq IVar(\rho)$ and that every logical variable occurring in Q has a type defined by ρ .

First we want to modify ρ by adding to it assignments to so-called *history* variables, i.e., auxiliary variables which record for every object its history, the sequence of communication records and activation records the object participates in. In languages like CSP such histories can be coded by integers: In CSP we can associate with each process an unique integer and thus code a communication record by an integer [Ap]. As there is no dynamic process creation in CSP a history is a sequence of communication records, which, given the coding of these records, can be coded too.

Given some coding of objects, it is not possible in our language to program an internal computation, using only auxiliary variables, which computes the code of an object. That is, we cannot program a mapping of histories into integers. Therefore to be able to prove completeness, using the technique applied to the proof theory of CSP, we have to extend our programming language. We do so by introducing for each $d \in C^+$ a new finite set of instance variables $IVar_d^c$, for an arbitrary c . It is not difficult to see how to code histories using these variables. However in the completeness proof we will not go into the details of coding these histories but simply assume to be given for each object of a class c_i defined by ρ a history variable h_i . For the details we refer to [AB]. We transform ρ to ρ' as follows:

Definition 7.1

Let \bar{x}^i be the instance variables occurring in D_i including the history variable h_i and \bar{y}^i be some new corresponding instance variables.

- Prefix every occurrence of an answer-statement, occurring in, say, D_i , by the multiple assignment $\bar{y}^i \Leftarrow \bar{x}^i$.
- Let the method m be declared in D_i as $S \uparrow e$; replace $S \uparrow e$ by

$$\bar{v} \Leftarrow \bar{y}^i; h_i \Leftarrow h_i \circ \langle m, \bar{u} \rangle; !; S; !'; \bar{y}^i \Leftarrow \bar{v}; h_i \Leftarrow h_i \circ \langle m, u_1, e \rangle \uparrow e,$$

where \bar{v} is a sequence of new temporary variables corresponding to the variables of the sequence \bar{y}^i , and \bar{u} are the formal parameters of m .

- Replace every occurrence of a statement $x \Leftarrow e_0!m(\bar{e})$ in, say, D_i , by

$$!; h_i \Leftarrow h_i \circ \langle m, e_0, \bar{e} \rangle; !'; x \Leftarrow e_0!m(\bar{e}); h_i \Leftarrow h_i \circ \langle m, \text{self}, x \rangle; !'.$$

- Replace every occurrence of a statement $x \leftarrow \text{new}$ in, say, D_i , by

$$!; x \leftarrow \text{new}; h_i \leftarrow h_i \circ < \text{self}, x >; !'$$

We assume the labels introduced to be distinct. It is important to keep in mind that the assignments to the history variables are in fact abbreviations of statements which compute the corresponding code. To be able to reason about invariance properties of an answer-statement we introduced some new instance variables \bar{y}^i to freeze the state just before the execution of the answer-statement. To ensure that after the execution of an answer-statement these variables still refer to the state just before the execution we assign the values of these variables to some new temporary variables when entering the body of a method. After the execution of a body of a method we then can recover the initial values of \bar{y}^i from these temporary variables. We assume that $\rho' = \langle U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} | c_n : S_n^{c_n} \rangle$, with $U_{m_1, \dots, m_k}^{c_1, \dots, c_{n-1}} = D_{1\bar{m}_1}^{c_1}, \dots, D_{n-1\bar{m}_{n-1}}^{c_{n-1}}$, where $D_{i\bar{m}_i}^{c_i} = \langle m_{1\bar{d}_1}^{i c_i} \Leftarrow \mu_{1\bar{d}_1}^{i c_i}, \dots, m_{n_i\bar{d}_{n_i}}^{i c_i} \Leftarrow \mu_{n_i\bar{d}_{n_i}}^{i c_i} \rangle : S_i^{c_i}$.

Definition 7.2

Let R denote an occurrence of a statement of D'_i , we define the set $\text{After}(R, D'_i)$ as follows: First, assume that R occurs in $S_i^{c_i}$, we put $\text{After}(R, D'_i) = \text{After}(R, S_i^{c_i})$, where $\text{After}(R, S)$ is defined as follows:

- If $R = S$ then $\text{After}(R, S) = \{E\}$
- If $S = S_1; S_2$
then $\text{After}(R, S) = \{R'; S_2 : R' \in \text{After}(R, S_1)\}$ if R occurs in S_1
 $\text{After}(R, S) = \text{After}(R, S_2)$ if R occurs in S_2
- If $S = \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}$ then $\text{After}(R, S) = \text{After}(R, S_1)$ if R occurs in S_1
 $\text{After}(R, S) = \text{After}(R, S_2)$ if R occurs in S_2
- If $S = \text{while } e \text{ do } S_1 \text{ od}$ then $\text{After}(R, S) = \{R'; S : R' \in \text{After}(R, S_1)\}$

Next, let R occur in S , S being the body of some method declared in D'_i , we define $\text{After}(R, D'_i)$ as follows:

$$\begin{aligned} \text{After}(R, D'_i) = \\ \bigcup \{ \text{After}(R_1, R'_1); \dots; \text{After}(R_n, R'_n) : \quad R_1 = R, R'_1 = S, R'_n = S_i^{c_i}, \forall 1 < i \leq n : \\ R_i = \text{answer}(\dots, m_i, \dots), m_i \Leftarrow R'_{i-1} \uparrow e_i \in D'_i \}. \end{aligned}$$

(Here, $X_1; \dots; X_n$, X_i being a set of statements, is defined by $\{R_1; \dots; R_n : R_i \in X_i\}$.)

Furthermore we define $Before(R, D'_i)$ as follows:

$$Before(R, D'_i) = \{R; R' : E; R' \in After(R, D'_i)\}.$$

Finally, for $R = x \leftarrow \text{wait}(m, e_0)$, associated with the send-statement $R' = x \leftarrow e_0!m(\bar{e})$, we define

$$Before(R, D'_i) = \{R; R'' : E; R'' \in After(R', D'_i)\}.$$

The intuition formalized by this definition should be clear: $After(R, D'_i)$ characterizes control when R has just been executed, while $Before(R, D'_i)$ characterizes control in those cases that R is about to be executed. The complication arising when R occurs in the body of some method is due to the fact that we have to take into account chains of answered methods of arbitrary length. We note that we assume some mechanism to distinguish between different occurrences of a statement.

Next we modify the precondition p of the valid correctness formula $\{p[z/self]\}_\rho\{Q\}$ as follows:

Definition 7.3

We define

$$\bar{p}^{c_n} = p^{c_n} \wedge \bigwedge_{x \in W} (x \doteq z_x) \wedge |h_n| \doteq 0,$$

where $W = IVar_{Int}^{c_n} \cup IVar_{Bool}^{c_n}$, z_x being a new logical variable uniquely associated with the instance variable x . These newly introduced variables are used to “freeze” that part of the initial state as specified by the integer and boolean variables.

Note that the assertion $|h_n| \doteq 0$ should be interpreted as an abbreviation of an assertion expressing the same fact, i.e., that there is no history yet, in terms of some particular coding of the histories.

To define the *expressibility* of some set of states we have the following definition:

Definition 7.4

For R occurring in, say, D'_i , we define

$$\begin{aligned} V(R) &= IVar(D'_i) \cup TVar(D'_i) && \text{if } R \text{ occurs in the body of some method} \\ &= IVar(D'_i) && \text{if } R \text{ occurs in } S_i^{c_i}. \end{aligned}$$

Furthermore, for $Y \subseteq IVar^c \cup TVar$, σ, σ' , and $\alpha \in \sigma^{(c)} \cap \sigma'^{(c)}$, we define

$$\sigma(\alpha) =_Y \sigma'(\alpha) \text{ iff } \sigma(\alpha)(v) = \sigma'(\alpha)(v), \text{ for all } v \in Y.$$

Each of the following lemmas state the expressibility of a set of states which collects all those states occurring during a particular computation whenever control is at some specific point.

Lemma 7.5

Let R be a substatement occurring in ρ' , say, R occurs in D'_i . There exists a local assertion $Pre(R)$ with $IVar(Pre(R)) \cup TVar(Pre(R)) \subseteq V(R)$, describing the local state of objects of class c_i , such that

$$\langle \alpha, \sigma \rangle, \omega \models Pre(R) \text{ iff}$$

- $\exists (X_0, \sigma_0) \rightarrow^* (X', \sigma')$, such that $Init_{\rho'}((X_0, \sigma_0))$,
- $X'(\alpha) \in Before(R, D'_i)$,
- $\langle \beta, \sigma_0 \rangle, \omega \models \bar{p}$, β being the root-object,
- $\sigma'(\alpha) =_{V(R)} \sigma(\alpha)$.

This local assertion $Pre(R)$ describes all the local states $\langle \alpha, \sigma \rangle$ for which there exists an intermediate configuration (X', σ') of a computation of ρ' , starting from an initial state of the root-object which satisfies \bar{p} , such that $\sigma'(\alpha)$ equals $\sigma(\alpha)$ with respect to the variables of $V(R)$, and furthermore, in the configuration (X', σ') the object α is about to execute R .

Using the techniques of [AB] one can show that the assertion $Pre(R)$ exists. Note that the set $Before(R, D'_i)$ (and $After(R, D'_i)$ as well) is recursive. (To decide if a statement occurs in, say, $After(R, D'_i)$, we only have to look at the sets $After(R_1, R'_1); \dots; After(R_n, R'_n)$, with n the length of the statement. But these sets are finite and there are only finitely many of them.)

Lemma 7.6

Let R be a substatement occurring in ρ' , say, R occurs in D'_i . There exists a local assertion $Post(R)$ with $IVar(Pre(R)) \cup TVar(Pre(R)) \subseteq V(R)$, describing the local state of objects of class c_i such that

$$\langle \alpha, \sigma \rangle, \omega \models Post(R) \text{ iff}$$

- $\exists (X_0, \sigma_0) \rightarrow^* (X', \sigma')$, such that $Init_{\rho'}((X_0, \sigma_0))$,
- $X'(\alpha) \in After(R, D'_i)$,
- $\langle \beta, \sigma_0 \rangle, \omega \models \bar{p}$, β being the root-object,

- $\sigma'(\alpha) =_{V(R)} \sigma(\alpha)$.

This local assertion describes all the local states $\langle \alpha, \sigma \rangle$ for which there exists an intermediate configuration (X', σ') of a computation of ρ' , starting from an initial state of the root-object which satisfies \bar{p} , such that $\sigma'(\alpha)$ equals $\sigma(\alpha)$ with respect to the variables of $V(R)$, and furthermore, in the configuration (X', σ') the object α has just finished executing R . The expressibility of this assertion is proved in the same way as the assertion $Pre(R)$.

Lemma 7.7

Again, let R be a substatement occurring in, say, D'_i . There exists a local assertion $Pre(R)_Y$ with $IVar(Pre(R)_Y) \cup TVar(Pre(R)_Y) \subseteq Y \cap V(R)$, where $Y \subseteq IVar^{c_i} \cup TVar$, such that

$$\langle \alpha, \sigma \rangle, \omega \models Pre(R)_Y \text{ iff}$$

there exists σ' such that $\sigma'(\alpha) =_Y \sigma(\alpha)$ and $\langle \alpha, \sigma' \rangle, \omega \models Pre(R)$.

The expressibility of this assertion is proved in the same way as the assertion $Pre(R)$.

Lemma 7.8

Let R be a substatement occurring in, say, D'_i , and p^{c_i} be such that $TVar(p) = \emptyset$. There exists a local assertion $SP(p, R)$ such that $TVar(SP(p, R)) = \emptyset$ and

$$\langle \alpha, \sigma \rangle, \omega \models SP(p, R) \text{ iff}$$

- $\exists (R, \theta_0) \rightarrow_L^* (E, \theta_1)$,
- $\theta_0, \omega \models p$,
- $\sigma_1(\alpha) =_{IVar(D'_i)} \sigma(\alpha)$, where $\theta_1 = \langle \alpha, \sigma_1 \rangle$.

The assertion $SP(p, R)$ expresses what is called the strongest postcondition of the statement R with respect to the precondition p . Note that $SP(p, R)$ specifies the behaviour of R only with respect to the instance variables of D'_i . The expressibility of this assertion is proved in the same way as the assertion $Pre(R)$. Next we have the following lemma:

Lemma 7.9

For every class c_i defined by ρ' there exists a local assertion $Lhist_i^{c_i}$, with $TVar(Lhist_i) = \emptyset$ and $IVar(Lhist_i) = \{h_i\}$, such that

$$\langle \alpha, \sigma \rangle, \omega \models Lhist_i \text{ iff}$$

- $\exists (X_0, \sigma_0) \rightarrow^* (X', \sigma'), \text{Init}_{\rho'}((X_0, \sigma_0)),$
- $\langle \beta, \sigma_0 \rangle, \omega \models \bar{p}, \beta$ being the root-object,
- $\sigma(\alpha)(h_i) = \sigma'(\alpha)(h_i).$

The expressibility of this assertion is proved among the same lines as the assertions defined above. The assertion *Lhist* holds in a local state $\langle \alpha, \sigma \rangle$ iff there exists an intermediate configuration (X', σ') of a computation of ρ' , starting from an initial state of the root-object which satisfies \bar{p} , such that σ and σ' agree with respect to the history of α . Finally, we have the following lemma,

Lemma 7.10

There exists a global assertion I such that $I\text{Var}(I) \subseteq \{h_1, \dots, h_n\}$ and

$$\sigma, \omega \models I \text{ iff}$$

- $\exists (X_0, \sigma_0) \rightarrow^* (X', \sigma'), \text{with } \text{Init}_{\rho'}((X_0, \sigma_0)),$
- X' is stable (see definition 6.11),
- $\langle \alpha, \sigma_0 \rangle, \omega \models \bar{p},$
- $\sigma'(\alpha)(h_i) = \sigma(\alpha)(h_i), \text{ for } \alpha \in \sigma'^{(c_i)}, c_i \in \{c_1, \dots, c_n\},$
- $\sigma^{(c_i)} = \sigma'^{(c_i)}, c_i \in \{c_1, \dots, c_n\}.$

This assertion I , which we call the global invariant, describes all states σ for which there exists an intermediate configuration (X', σ') of a computation of ρ' , starting from an initial state of the root-object which satisfies \bar{p} , such that σ and σ' agree with respect to the existing objects and with respect to the histories of these objects. Note that as X' is stable we have that the history recorded by an existing object of σ' equals the history obtained from h by deleting from it all the communication and activation records not involving this object. The expressibility of this assertion is shown along the lines of [AB].

Using the above lemmas we now define the set of assumptions and commitments for each class $c_i \in \{c_1, \dots, c_n\}$.

Definition 7.11

We define

- $A_i = \{l'.\text{Post}(R) : R = l; R'; l' \in D'_i, \text{ with } R' \text{ containing a new-statement}\} \cup$
 $\{l'.\text{Post}(R) : R = l; R'; l' \in D'_i, \text{ with } R' \text{ containing a send-statement}\} \cup$
 $\{l.\text{Post}(R) : m \Leftarrow R; l; S; l'; R' \uparrow e \in D'_i\} \cup$
 $\{l'.\text{SP}(\bar{y}^i \doteq \bar{x}^i, \text{answer}(\bar{m})) \wedge \text{Lhist}_i : l; \text{answer}(\bar{m}); l' \in D'_i\}$

- $C_i = \{ \text{!}.Pre(R) : R = \text{!}; R'; l' \in D'_i, \text{ with } R' \text{ containing a new-statement} \} \cup$
- $\{ \text{!}.Pre(R) : R = \text{!}; R'; l' \in D'_i, \text{ with } R' \text{ containing a send-statement} \} \cup$
- $\{ \text{!}.Pre(R)_{IVar} : R = \text{!}; \text{answer}(\bar{m}); l' \in D'_i \} \cup$
- $\{ \text{!}.Pre(R)_{ITVar \setminus \{x\}} : R = x \leftarrow \text{wait}(m, e_0), \text{ with } \text{!}; x \leftarrow e_0!m(\bar{e}) \in D'_i \}$

Here $R \in D'_i$ should be interpreted to mean that R occurs in D'_i .

It is important to note that given a statement $\text{!}; \text{answer}(\bar{m}); l'$ we did not associate the assertion $Post(\text{!}; \text{answer}(\bar{m}); l')$ with the label l' . The reason for this being that this assumption cannot be justified in the cooperation test, because when exiting the body of one of the methods of \bar{m} we have no information about the particular answer-statement which gave rise to the execution of this method. However we will see in the following lemma how we can strengthen the assumption $SP(\bar{y}^i \doteq \bar{x}^i, \text{answer}(\bar{m})) \wedge Lhist_i$ by some reasoning within the local proof system to the assertion $Post(\text{!}; \text{answer}(\bar{m}); l')$. We are now ready for the following lemma:

Lemma 7.12

We have

$$\vdash (A_i, C_i : \{ Pre(R) \} R \{ Post(R) \}),$$

where R occurs in D'_i such that R is normal, i.e., R does not occur in a bracketed section.

Proof

The proof proceeds by induction on the structure of R . We treat the case $R = \text{!}; \text{answer}(m_1, \dots, m_n); l'$, the other cases being straightforward.

Let m_j be declared as $R'_j; l'_j; R_j; l''_j; R''_j \uparrow e_j$, furthermore, let $p = Pre(\text{!}; \text{answer}(m_1, \dots, m_n); l')$ and $p' = p[\bar{y}^i / \bar{x}^i]$. We first show that

$$\vdash (\emptyset, \emptyset : \{ p'[\bar{y}/\bar{u}] \} R'_j; R_j; R''_j \{ p'[\bar{y}/\bar{u}] \}),$$

where $TVar(p') = \bar{u}$ and \bar{y} are some new instance variables:

1. $(\emptyset, \emptyset : \{ p'[\bar{y}/\bar{u}][\bar{v}/\bar{y}^i] \} R''_j \{ p'[\bar{y}/\bar{u}] \})$, by (LIASS).
2. $(\emptyset, \emptyset : \{ p'[\bar{y}/\bar{u}][\bar{v}/\bar{y}^i] \} R_j \{ p'[\bar{y}/\bar{u}][\bar{v}/\bar{y}^i] \})$, by (INV).
3. $(\emptyset, \emptyset : \{ p'[\bar{y}/\bar{u}] \} R'_j \{ p'[\bar{y}/\bar{u}][\bar{v}/\bar{y}^i] \})$, by (LTASS).

So applying the rule (BA) gives us

$$\vdash (A_i, C_i : \{ p' \wedge p_1 \} \text{!}; \text{answer}(m_1, \dots, m_n); l' \{ p' \wedge q_1 \}),$$

where

- $p_1 = C(l) = \text{Pre}(l; \text{answer}(m_1, \dots, m_n); l')_{IVar}$,
- $q_1 = A(l') = \text{SP}(\bar{y}^i \doteq \bar{x}^i, \text{answer}(m_1, \dots, m_n)) \wedge \text{Lhist}_i$.

It is not difficult to check that

$$\models \text{Pre}(l; \text{answer}(m_1, \dots, m_n); l') \rightarrow p' \wedge p_1.$$

(Note that $\models \text{Pre}(l; \text{answer}(m_1, \dots, m_n); l') \rightarrow \bar{y}^i \doteq \bar{x}^i$.) So applying the consequence rule gives us

$$\vdash (A_i, C_i : \{\text{Pre}(l; \text{answer}(m_1, \dots, m_n); l')\}l; \text{answer}(m_1, \dots, m_n); l' \{p' \wedge q_1\}).$$

For an application of the consequence rule it thus suffices to show that the following implication holds

$$p' \wedge q_1 \rightarrow \text{Post}(l; \text{answer}(m_1, \dots, m_n); l').$$

Here we go. Let

$$\langle \alpha, \sigma \rangle, \omega \models p' \wedge q_1.$$

From $\langle \alpha, \sigma \rangle, \omega \models \text{Lhist}_i$ it follows that there exists a computation

$$(X_0, \sigma_0) \rightarrow^* (X_m, \sigma_m)$$

of ρ' such that $\sigma_m(\alpha)(h_i) = \sigma(\alpha)(h_i)$. Furthermore, we may assume without loss of generality that α has just finished executing a bracketed section.

Let $\sigma' = \sigma\{\sigma(\alpha)(\bar{y}^i)/\alpha, \bar{x}^i\}$. By $\langle \alpha, \sigma' \rangle, \omega \models p$ it then follows that there exists a computation

$$(X_0, \sigma_0) \rightarrow^* (X_n, \sigma_n)$$

of ρ' such that $X_n(\alpha) \in \text{Before}(l; \text{answer}(m_1, \dots, m_n); l', D'_i)$ and $\sigma_n(\alpha) =_{V(R)} \sigma'(\alpha)$ ($R = l; \text{answer}(m_1, \dots, m_n); l'$). Note that we may assume that this computation starts from the same initial configuration (X_0, σ_0) as the computation which exists according to $\sigma, \omega \models \text{Lhist}_i$ because of the use of freeze variables in definition 7.3.

Finally, by $\langle \alpha, \sigma \rangle, \omega \models q_1$ we have for some history h'

$$(\text{answer}(m_1, \dots, m_n), \theta) \xrightarrow{h'} (E, \theta'),$$

with $\theta_{(1)} = \theta'_{(1)} = \alpha$, $\theta'_{(2)}(\alpha) =_{IVar(D'_i)} \sigma(\alpha)$, and $\theta, \omega \models \bar{y}^i \doteq \bar{x}^i$.

As $\sigma_m(\alpha)(h_i) = \sigma(\alpha)(h_i) = \theta'_{(2)}(\alpha)(h_i)$ we have $\theta'_{(2)}(\alpha)(h_i) = \sigma_m(\alpha)(h_i) = h'' \circ h'$ for some history h'' . Now let

$$(X_0, \sigma_0) \rightarrow^* (X_k, \sigma_k) \rightarrow^* (X_m, \sigma_m)$$

such that $\sigma_k(\alpha)(h_i) = h''$ and α is about to execute a bracketed section. Now as $\theta'_{(2)}(\alpha)(h_i) = \theta_{(2)}(\alpha)(h_i) \circ h' = h'' \circ h'$ we have

$$\theta_{(2)}(\alpha)(h_i) = h'' = \sigma_k(\alpha)(h_i). \quad (7.1)$$

From

$$\begin{aligned} \theta_{(2)}(\alpha)(\bar{y}^i) &= \\ \theta'_{(2)}(\alpha)(\bar{y}^i) &= \\ \sigma(\alpha)(\bar{y}^i) &= \\ \sigma'(\alpha)(\bar{x}^i) &= \\ \sigma_n(\alpha)(\bar{x}^i) & \end{aligned}$$

and

$$\theta, \omega \models \bar{y}^i \doteq \bar{x}^i$$

it follows that

$$\theta_{(2)}(\alpha)(h_i) = \sigma_n(\alpha)(h_i). \quad (7.2)$$

So from 7.1 and 7.2 we infer that $\sigma_n(\alpha)(h_i) = \sigma_k(\alpha)(h_i)$. From this in turn we derive that $X_k(\alpha) = X_n(\alpha)$ and $\sigma_k(\alpha) = \sigma_n(\alpha)$. (Note that the behaviour of an object is uniquely determined with respect to the behaviour of the environment.) Now, from $\sigma_k(\alpha) = \sigma_n(\alpha)$ (using the above sequence of identities) it follows that $\sigma_k(\alpha) =_{IVar(D'_i)} \theta_{(2)}(\alpha)$. From which it is not difficult to derive that $X_m(\alpha) \in \text{After}(\text{!}; \text{answer}(m_1, \dots, m_n); l', D'_i)$ and $\sigma_m(\alpha) =_{IVar(D'_i)} \theta'_{(2)}(\alpha) =_{IVar(D'_i)} \sigma(\alpha)$ (use again that the behaviour of an object is uniquely determined with respect to the behaviour of the environment). Furthermore, we have

$$\begin{aligned} \sigma_m(\alpha)(u) &= \\ \sigma_k(\alpha)(u) &= \\ \sigma_n(\alpha)(u) &= \\ \sigma'(\alpha)(u) &= \\ \sigma(\alpha)(u), & \end{aligned}$$

where $u \in V(R)$ is a temporary variable. Note that the first identity follows from our assumption that α in (X_m, σ_m) has just finished executing a bracketed section, because then, as $\sigma_m(\alpha)(h_i) = \sigma(\alpha)(h_i) = \theta'_{(2)}(\alpha)(h_i)$, we have that α (in (X_m, σ_m)) has just finished executing $\text{!}; \text{answer}(m_1, \dots, m_n); l'$. Thus we conclude that $\sigma(\alpha) =_{V(R)} \sigma_m(\alpha)$, so

$$\langle \alpha, \sigma \rangle, \omega \models \text{Post}(\text{!}; \text{answer}(m_1, \dots, m_n); l').$$

□

In the following lemmas we show that the other requirements of the cooperation test are satisfied.

Lemma 7.13

Let $l; \text{answer}(\dots, m, \dots); l'$ occur in D'_i , with m declared as $R_1; l_1; R; l_2; R_2 \uparrow e$. Furthermore, let $l'_1; R'_1; \bar{l}; x \leftarrow e_0!m(e_1, \dots, e_n); R'_2; l'_2$ occur in D'_j . Then the following intermediate correctness formulas are valid:

$$\begin{aligned} & \{I \wedge p'_1[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge p[z_2/\text{self}] \wedge P\} \\ & \quad (z_1, R'_1[\bar{y}_1/\bar{u}_1]) \parallel (z_2, R_1[\bar{y}_2/\bar{u}_2]) \\ & \{I \wedge \bar{p}[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge p_1[\bar{y}_2/\bar{u}_2][z_2/\text{self}]\} \end{aligned}$$

and

$$\begin{aligned} & \{I \wedge \bar{p}[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge p_2[\bar{y}_2/\bar{u}_2][z_2/\text{self}] \wedge Q\} \\ & \quad (z_1, R'_2[\bar{y}_1/\bar{u}_1]) \parallel (z_2, R_2[\bar{y}_2/\bar{u}_2]) \\ & \{I \wedge p'_2[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge p'[\bar{y}_2/\bar{u}_2][z_2/\text{self}]\} \end{aligned}$$

where

- $p = C_i(l) = \text{Pre}(l; \text{answer}(\dots, m, \dots); l')_{IVar}$,
- $p' = A_i(l') = SP(\bar{y}^i \doteq \bar{x}^i, \text{answer}(\dots, m, \dots)) \wedge Lhist_i$,
- $p_1 = A_i(l_1) = \text{Post}(R_1)$, $p_2 = C_i(l_2) = \text{Pre}(R_2)$,
- $p'_1 = C_j(l'_1) = \text{Pre}(l'_1; R'_1; \bar{l}; x \leftarrow e_0!m(e_1, \dots, e_n); R'_2; l'_2)$,
- $\bar{p} = C_j(\bar{l}) = \text{Pre}(x \leftarrow \text{wait}(m, e_0))_{ITvar \setminus \{x\}}$,
- $p'_2 = A_j(l'_2) = \text{Post}(l'_2; R'_2; \bar{l}; x \leftarrow e_0!m(e_1, \dots, e_n); R'_2; l'_2)$,
- $P = e_0[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \doteq z_2 \wedge \bigwedge_i e_i[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \doteq z_2.y'_i \wedge \bigwedge_j z_2.y''_j \doteq \text{nil}$,
- $Q = z_1 \doteq z_2.y'_1 \wedge z_1.x \doteq e[\bar{y}_2/\bar{u}_2][z_2/\text{self}]$.

Here $\bar{y}' \cup \bar{y}'' = \bar{y}_2$, with \bar{y}' being the instance variables corresponding to the formal parameters and \bar{y}'' corresponding to the local variables of m .

Proof

We start with the proof of the validity of the first correctness formula.

Let

$$\sigma, \omega \models I \wedge p'_1[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge p[z_2/\text{self}] \wedge P.$$

Furthermore let

$$(\sigma, (\alpha_1, R'_1[\bar{y}_1/\bar{u}_1]), (\alpha_2, R_1[\bar{y}_2/\bar{u}_2])) \rightarrow^* (\sigma', (\alpha_1, E), (\alpha_2, E)),$$

where $\alpha_1 = \omega(z_1)$ and $\alpha_2 = \omega(z_2)$. Next we define $\sigma'' = \sigma\{\sigma(\alpha_1)(\bar{y}_1)/\alpha_1, \bar{u}_1\}$. By $\sigma'', \omega \models I$, $\langle \alpha_1, \sigma'' \rangle, \omega \models p'_1$ and $\langle \alpha_2, \sigma'' \rangle, \omega \models p$ it then follows that there exists a computation of ρ'

$$(X_0, \sigma_0) \rightarrow^* (X_n, \sigma_n)$$

such that

- $X_n(\alpha_1) \in \text{Before}(R'_1, D'_j)$ and $X_n(\alpha_2) \in \text{Before}(\text{!}; \text{answer}(\dots, m, \dots); l', D'_i)$,
- $\sigma_n(\alpha_1) =_{V(R)} \sigma''(\alpha_1)$ ($R = R'_1; x \leftarrow e_0!m(e_1, \dots, e_n); R'_2$) and $\sigma_n(\alpha_2) =_{IVar(D'_i)} \sigma''(\alpha_2)$,
- $\sigma_n^{(c)} = \sigma''^{(c)}$, $c \in \{c_1, \dots, c_n\}$,
- $\sigma_n(\alpha)(h_i) = \sigma''(\alpha)(h_i)$, $\alpha \in \sigma''^{(c_i)}$, $c_i \in \{c_1, \dots, c_n\}$.

Note that, as the history of α_1 (α_2) in the computation which exists according to $\langle \alpha_1, \sigma'' \rangle, \omega \models p'_1$ ($\langle \alpha_2, \sigma'' \rangle, \omega \models p$) equals that of α_1 (α_2) in the computation which exists according to $\sigma'', \omega \models I$, we have that in this latter computation α_1 is about to execute R'_1 (and α_2 is about to execute $\text{!}; \text{answer}(\dots, m, \dots); l'$) and the local states of α_1 (α_2) in the final states of these computations coincide. (Again, the above observation is based on the fact that the behaviour of an object is completely determined given its interactions with the environment, furthermore we make use of that the computations which exists according to $\sigma'', \omega \models I$, $\langle \alpha_1, \sigma'' \rangle, \omega \models p'_1$ and $\langle \alpha_2, \sigma'' \rangle, \omega \models p$ all start from the same initial state, which is due to the use of the logical variables z_x , introduced in definition 7.3, as freeze variables.)

Now let

$$(X_n, \sigma_n) \rightarrow^* (X_k, \sigma_k)$$

such that

- $X_k(\alpha_1) \in \text{After}(R'_1, D'_j)$ and $X_k(\alpha_2) \in \text{Before}(x \leftarrow \text{wait}(m, e_0), D'_i)$,
- from $\langle X_n, \sigma_n \rangle$ to $\langle X_k, \sigma_k \rangle$ only α_1 and α_2 are executing; α is executing R'_1 and α_2 is executing R_2 .

Note that such a computation exists because $\sigma'', \omega \models e_0[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \doteq z_2$, so we have $\mathcal{E}[\![e_0]\!](\langle \alpha_1, \sigma_n \rangle) = \mathcal{E}[\![e_0]\!](\langle \alpha_1, \sigma'' \rangle) = \alpha_2$.

It is not difficult to see that

$$\sigma_k(\alpha_1)(\bar{u}_1) = \sigma'(\alpha_1)(\bar{y}_1),$$

$$\begin{aligned}
\sigma_k(\alpha_2)(\bar{u}_2) &= \sigma'(\alpha_2)(\bar{y}_2), \\
\sigma_k(\alpha_1)(x) &= \sigma'(\alpha_1)(x), \quad x \in IVar(D'_j), \\
\sigma_k(\alpha_2)(x) &= \sigma'(\alpha_2)(x), \quad x \in IVar(D'_i).
\end{aligned}$$

Furthermore we have

$$\sigma_k^{(c)} = \sigma'^{(c)}, \quad c \in \{c_1, \dots, c_n\}$$

and

$$\sigma_k(\alpha)(h_i) = \sigma'(\alpha)(h_i), \quad \alpha \in \sigma_k^{(c_i)}, \quad c_i \in \{c_1, \dots, c_n\}.$$

So we conclude that

$$\begin{aligned}
\sigma', \omega &\models I, \\
\langle \alpha_1, \sigma' \rangle, \omega &\models Pre(x \leftarrow \text{wait}(m, e_0))[\bar{y}_1/\bar{u}_1], \\
\langle \alpha_2, \sigma' \rangle, \omega &\models Post(R_1)[\bar{y}_2/\bar{u}_2].
\end{aligned}$$

Now it is not difficult to see that

$$\models Pre(x \leftarrow \text{wait}(m, e_0)) \rightarrow \bar{p}.$$

So we conclude

$$\sigma', \omega \models I \wedge \bar{p}[\bar{y}_1/\bar{u}_1][z_1/\text{self}] \wedge p_1[\bar{y}_2/\bar{u}_2][z_2/\text{self}].$$

The proof of the validity of the second correctness formula is similar. \square

Lemma 7.14

Let $l; x \leftarrow \text{new}; R_2; l'$ occur in D'_i . Then the following correctness assertion is valid:

$$\begin{aligned}
&\{I \wedge p[\bar{y}/\bar{u}][z/\text{self}]\} \\
&(z, (x \leftarrow \text{new}; R_2)[\bar{y}/\bar{u}]) \\
&\{I \wedge p'[\bar{y}/\bar{u}][z/\text{self}] \wedge q[z.x/\text{self}]\}
\end{aligned}$$

where

- $p = Pre(l; x \leftarrow \text{new}; R_2; l')$, $p' = Post(l; x \leftarrow \text{new}; R_2; l')$, and $q = Pre(S_j^{c_j})$, assuming the type of x to be c_j ,
- \bar{y} are some new instance variables corresponding to the temporary variables \bar{u} .

Proof

Let

$$\sigma, \omega \models I \wedge p[\bar{y}/\bar{u}][z/\text{self}],$$

and

$$(\sigma, (\alpha, (x \leftarrow \text{new}; R_2)[\bar{y}/\bar{u}])) \rightarrow^* (\sigma', (\alpha, E)),$$

where $\alpha = \omega(z)$. Let $\sigma'' = \sigma\{\sigma(\alpha)(\bar{y})/\bar{u}\}$. It follows that there exists a computation of ρ'

$$(X_0, \sigma_0) \rightarrow^* (X_n, \sigma_n)$$

(the existence of such a computation is justified as in the proof of the previous lemma) such that

- $X_n(\alpha) \in \text{Before}(l; x \leftarrow \text{new}; R_2; l', D'_i)$,
- $\sigma_n(\alpha) =_{V(R)} \sigma''(\alpha)$, $R = x \leftarrow \text{new}; R_2$,
- $\sigma_n^{(c)} = \sigma''^{(c)}$, $c \in \{c_1, \dots, c_n\}$,
- $\sigma_n(\beta)(h_i) = \sigma''(\beta)(h_i)$, $\beta \in \sigma''^{(c_i)}$, $c_i \in \{c_1, \dots, c_n\}$.

Next let

$$(X_n, \sigma_n) \rightarrow^* (X_k, \sigma_k)$$

such that $X_k(\alpha) \in \text{After}(l; x \leftarrow \text{new}; R_2; l', D'_i)$ and from (X_n, σ_n) to (X_k, σ_k) only α is executing. Now from

$$\sigma_k(\beta) = \sigma'(\beta),$$

with β the newly created object,

$$\sigma_k(\alpha)(\bar{u}) = \sigma'(\alpha)(\bar{y}),$$

$$\sigma_k(\alpha)(x) = \sigma'(\alpha)(x), \quad x \in \text{IVar}(D'_i),$$

and, finally,

$$\langle \beta, \sigma_k \rangle, \omega \models \text{Pre}(S_j^{c_j})$$

it follows that

$$\sigma', \omega \models I \wedge p'[\bar{y}/\bar{u}][z/\text{self}] \wedge q[z.x/\text{self}].$$

□

Theorem 7.15

The following formula about ρ' , which is called the most general correctness formula about ρ' , is derivable:

$$\begin{aligned} & \{ \text{Pre}(S_n^{c_n})[z_n/\text{self}] \} \\ & \rho' \\ & \{ I \wedge \bigwedge_{i \neq n} \forall z_i \text{Post}(S_i^{c_i})[z_i/\text{self}] \wedge \text{Post}(S_n^{c_n})[z_n/\text{self}] \}. \end{aligned}$$

Proof

From lemma 7.12 it follows that

$$\vdash (A_i, C_i : \{Pre(S_i^{c_i})\} S_i^{c_i} \{Post(S_i^{c_i})\}),$$

and

$$\vdash (A_i, C_i : \{A_i(l)\} R \{C_i(l')\}),$$

where R occurs in the body $R_1; l; R; l'. R_2 \uparrow e$ of some method m defined in D'_i . Furthermore it is not difficult to prove that

$$\models Pre(S_n^{c_n})[z_n/self] \wedge \forall z'(z' \doteq z_n) \wedge \bigwedge_{1 \leq i < n} (\forall z_i \text{ false}) \rightarrow I.$$

(The variables z_i are assumed to be of type c_i .) From the completeness of the intermediate proof system and the above lemmas it follows that the cooperation test holds. (The completeness of the intermediate proof system is proved in a similar way as the completeness of the usual Hoare-style proof system for sequential programs.) An application of the rule (PR) then finishes the proof. \square

We are now ready for the completeness theorem.

Theorem 7.16

Every valid correctness formula $\{p[z_{c_n}/self]\} \rho \{Q\}$ is derivable.

$$\vdash \{p[z_{c_n}/self]\} \rho \{Q\}.$$

Proof

By the previous theorem we have the derivability of the correctness formula

$$\begin{array}{c} \{Pre(S_n^{c_n})[z_n/self]\} \\ \rho' \\ \{I \wedge \bigwedge_{i \neq n} \forall z_i Post(S_i^{c_i})[z_i/self] \wedge Post(S_n^{c_n})[z_n/self]\}. \end{array}$$

It is not difficult to prove that $\models p^{c_n} \wedge \bigwedge_{x \in W} x \doteq \text{nil} \rightarrow Pre(S_n^{c_n})$, where $W = \bigcup_c IVar_c^{c_n}$. Furthermore, as $I \wedge \bigwedge_{i \neq n} \forall z_i Post(S_i^{c_i})[z_i/self] \wedge Post(S_n^{c_n})[z_n/self]$ can be easily seen to characterize precisely the set of final states, we have

$$\models I \wedge \bigwedge_{i \neq n} \forall z_i Post(S_i^{c_i})[z_i/self] \wedge Post(S_n^{c_n})[z_n/self] \rightarrow Q.$$

By the consequence rule and the rule (INIT) we thus have

$$\vdash \{p[z_n/self]\} \rho' \{Q\}.$$

Applying the substitution rules (S1) and (S2), substituting every logical variable z_x by the corresponding instance variable x , and substituting every history variable by the empty sequence, denoted by nil , then gives us, after a trivial application of the consequence rule,

$$\vdash \{p[z_n/\text{self}]\}\rho'\{Q\}.$$

An application of the rule (AUX) then finishes the proof. \square

8 Conclusion

We have developed a formal proof system for reasoning about the partial correctness of programs written in the language POOL. We proved the system to be sound and (relative) complete with respect to a formal semantics.

We mention the following topics for future research: First, we have the problem of *compositionality*, i.e., the development of a proof system along the lines of [ZREB] in which the history variables introduced in the completeness proof as auxiliary variables are incorporated in the system itself.

Another interesting subject is the problem how to formalize reasoning about *deadlock* behaviour. Due to the presence of dynamic object creation the standard techniques developed for languages like CSP do not apply.

Finally, in the full language POOL an object can call its own methods. We did not study this feature because we wanted to focus on the remote procedure call mechanism in POOL. But we think we can incorporate the proof theory for recursive procedures ([Ap2]) in our assumption/commitment formalism.

Acknowledgements We are much indebted to the work of Pierre America on the proof theory for the language SPOOL. Furthermore we thank the members of the Amsterdam Concurrency Group, J.W. de Bakker, A. de Bruin, P.M.W. Knijnenburg, J.N. Kok, J.J.M.M. Rutten, E. de Vink en J.H.A. Warmerdam for their fruitfull comments.

References

- [AB] P. America, F.S. de Boer: *A proof system for a parallel programming language with dynamic process creation*. Technical Report, Technical University Eindhoven.

- [AB2] P. America, F.S. de Boer: *A proof theory for a sequential version of POOL*, Technical Report, Technical University Eindhoven.
- [Am] P. America: *Definition of the programming language POOL-T*. ESPRIT project 415A, Doc. No. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, September 1985.
- [Ap] K.R. Apt: *Formal justification of a proof system for CSP*. Journal ACM, Vol. 30, No. 1, 1983, pp. 197–216.
- [Ap2] K.R. Apt: *Ten years of Hoare logic: a survey-part 1*. Journal ACM, Vol. 3, No. 4, 1981, pp. 431–483.
- [AFR] K.R. Apt, N. Francez, W.P. de Roever: *A proof system for communicating processes*, ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, 1980, pp. 359–385.
- [Ba] J.W. de Bakker: *Mathematical theory of program correctness*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.
- [GR] R. Gerth, W.P. de Roever: *A proof system for concurrent Ada programs*. Science of Computer Programming 4, pp. 159–204.
- [Go] Adele Goldberg, David Robson: *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, 1983.
- [HR] J. Hooman, W.P. de Roever: *The quest goes on: towards compositional proof systems for CSP*. J.W. de Bakker, W.P. de Roever, G. Rozenberg (eds.): Current trends in concurrency, Proc. LPC/ESPRIT Advanced School, Springer Lecture Notes in Computer SCIENCE, Vol. 224, 1986.
- [ZREB] J. Zwiers, W.P. de Roever, P. van Emde Boas: *Compositionality and concurrent networks: soundness and completeness of a proof system*. Proceedings of the twelfth International Colloquium on Automata, Languages and Programming, Nafplion, Greece, July 15–19, 1985, Springer Lecture Notes in Computer Science 194, pp. 509–519.

Samenvatting

Dit proefschrift bestaat uit een inleiding en drie artikelen.

In de inleiding wordt een schets gegeven van de algemene problematiek van de korrektheid van programmas, met name wordt er ingegaan op de korrektheid van programma's geschreven in de parallelle object georiënteerde taal POOL.

In dit proefschrift wordt de korrektheidsproblematiek benaderd vanuit het gezichtspunt dat een programma op te vatten is als een toestandstransformatie. De specificatie van de korrektheid van een programma bestaat in deze optiek uit een "preconditie" en een "postconditie", asserties geformuleerd in één of andere logische taal. De betekenis van een dergelijke specificatie is dat elke terminerende executie van het betreffende programma in een toestand die voldoet aan de preconditie termineert in een toestand waarin de postconditie geldt.

In het eerste hoofdstuk wordt de bewijstheorie bestudeerd van een sequentiële versie van de taal POOL geheten SPOOL. Er wordt van het parallellisme in de taal POOL afgezien om de aandacht te vestigen op het probleem hoe te redeneren over het fenomeen van dynamisch zich ontwikkelende topologieën. In dit hoofdstuk wordt een bewijssysteem ontworpen waarin over dergelijke topologieën kan worden geredeneerd op een zelfde abstractieniveau als gegeven door de programmeertaal. De basisideeën die aan dit bewijssysteem ten grondslag liggen zijn ook van toepassing op de formalisering van het redeneren over de korrektheid van manipulaties op datastructuren als "records" (en "pointers" naar records) zoals deze voorkomen in talen als PASCAL.

In het tweede hoofdstuk wordt de formalisering van het redeneren over dynamisch zich ontwikkelende topologieën zoals beschreven in het eerste hoofdstuk toegepast op een parallelle versie van de taal POOL, de taal *P*. Door middel van deze taal worden de bewijstheoretische aspecten van het parallellisme van de taal POOL onderzocht. Hier toe wordt afgezien van het specifieke communicatiemechanisme van de taal POOL; in POOL communiceren objecten door middel van een "rendezvous" mechanisme. In de taal *P* daarentegen communiceren objecten door het versturen van waarden op een wijze zoals beschreven door CSP.

De bewijsmethode voor de parallelle taal *P* bestaat uit drie niveaus:

Het locale niveau Op dit niveau wordt over het interne gedrag van een object geredeneerd. De interactie van een object met de omgeving (de andere objecten) wordt op dit niveau beschreven door *aannames* over het gedrag van de omgeving.

De coöperatietest Op dit niveau wordt gecontroleerd of de aannames die geïntroduceerd zijn op het locale niveau onderling consistent zijn. Daartoe dient te worden geredeneerd over een systeem van objecten en de interactie tussen objecten.

Het globale niveau Op dit niveau wordt uiteindelijk beschreven hoe de locale specificaties van objecten gecombineerd kunnen worden tot een specificatie van het totale programma.

Uiteindelijk worden in het laatste hoofdstuk de resultaten van de voorgaande hoofdstukken toegepast op de taal POOL en uitgebreid tot een axiomatisering van het rendezvous mechanisme.

In elk hoofdstuk wordt de gezondheid ('soundness') en volledigheid van het betreffende bewijs systeem bewezen met betrekking tot een formele semantiek.

STELLINGEN

1. Dynamische procescreatie is bewijstheoretisch te karakteriseren door middel van een generalisatie van de technieken ontwikkeld voor talen als CSP (K.R. Apt, N. Francez, W.P. de Roever: A proof system for communicating processes, ACM Transactions on Programming Languages and Systems, Vol. 2, 1980, pp 359-483). Zie dit proefschrift.
2. De basisideeën die ten grondslag liggen aan de bewijstheorie van dynamische processtructuren zijn ook van toepassing op het redeneren op een zelfde abstractieniveau als in de taal PASCAL over operaties op datastructuren als "records".
3. De volledigheid van het bewijssysteem voor de taal POOL gebaseerd op de assertietaal waarin eigenschappen van dynamische processtructuren beschreven worden door middel van recursieve predikaten kan niet bewezen worden gebruik makend van de standaardtechnieken.
4. Het gedrag van een systeem van objecten kan volledig beschreven worden in termen van het locale gedrag en de interface van de objecten. Zie F.S. de Boer: A compositional proof system for dynamic process creation, technisch rapport Technische Universiteit Eindhoven, te verschijnen.
5. Een volledig abstracte semantiek voor parallele talen waarin processen asynchroon communiceren door middel van een gemeenschappelijke datastructuur kan gegeven worden door middel van maximale traces. Zie F.S. de Boer, J.N. Kok, C. Palamidessi en J.J.M.M. Rutten: The failure of failures: Towards a paradigm for asynchronous communication, technisch rapport CWI, Amsterdam, te verschijnen.
6. Met betrekking tot parallele talen lijkt het alleszins redelijk de notie van inbedding zoals voorgesteld door E. Shapiro aan te scherpen door additioneel de compositionaliteit van de compiler en de terminatieinvariantie van de decoder te eisen. Zie F.S. de Boer en C. Palamidessi: Concurrent logic languages: asynchronism and language comparison, Proceedings of the North American Conference on Logic Programming, Series in Logic Programming. The MIT Press, 1990.
7. De formalisatie van het redeneren over de totale korrektheid van programma's vereist meer zorg en aandacht dan algemeen werd verondersteld. Zie P. America en F.S. de Boer: Proving total correctness of recursive procedures, Information and Computation, Vol. 84, pp. 129-162, 1990.
8. Zonder de mogelijkheid in de assertietaal expliciet te refereren naar een specifieke incarnatie van een procedure is het redeneren over het monitorconcept onvolledig. Zie R. Gerth: Syntax-directed verification of distributed systems, proefschrift, R.U. Utrecht, 1989.
9. Voor de beschrijving van sequentiële compositie in temporele logica is de "chop" operator niet noodzakelijk. Zie F.S. de Boer: Compositionality in temporal logic, Future Generation Computer Systems, Vol. 6, pp 287-299, 1990.
10. De ontwikkeling van een algemene theorie over de expressiviteit van (parallele) programmeer talen vormt een veelbelovend nieuw onderzoeksgebied.