Journal of Computer and System Sciences 78 (2012) 823–852

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Verification of object-oriented programs: A transformational approach

Krzysztof R. Apt [a,b,*], Frank S. de Boer [a,c], Ernst-Rüdiger Olderog [d], Stijn de Gouw [a,c]

[a] Centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands
[b] University of Amsterdam, Institute of Language, Logic and Computation, Amsterdam, The Netherlands
[c] Leiden Institute of Advanced Computer Science, University of Leiden, The Netherlands
[d] Department of Computing Science, University of Oldenburg, Germany

ARTICLE INFO

Article history:
Received 21 April 2010
Received in revised form 16 February 2011
Accepted 5 August 2011
Available online 25 August 2011

Keywords:
Object-oriented programs
Null references
Aliasing
Inheritance
Subtyping
Syntax-directed transformation
Recursive programs
Program verification
Strong partial correctness
Relative completeness

ABSTRACT

We show that verification of object-oriented programs by means of the assertional method can be achieved in a simple way by exploiting a syntax-directed transformation from object-oriented programs to recursive programs. This transformation suggests natural proofs rules and its correctness helps us to establish soundness and relative completeness of the proposed proof system. One of the difficulties is how to properly deal in the assertion language with the instance variables and aliasing. The discussed programming language supports arrays, instance variables, failures and recursive methods with parameters. We also explain how the transformational approach can be extended to deal with other features of object-oriented programming, like classes, inheritance, subtyping and dynamic binding.

1. Introduction

1.1. Background and motivation

Ever since its introduction in [14] the assertional method has been one of the main approaches to program verification. Initially proposed for the modest class of **while** programs, it has been extended to several more realistic classes of programs, including recursive programs (starting with [15]), programs with nested procedure declarations (see [19]), parallel programs (starting with [23] and [24]), and distributed programs based on synchronous communication (see [4]). At the same time research on the theoretical underpinnings of the proposed proof systems resulted in the introduction in [10] of the notion of relative completeness and in the identification of the inherent incompleteness for a comprehensive ALGOL-like programming language (see [9]).

However, (relative) completeness of proof systems proposed for current object-oriented programming languages (see the related work section below) remained largely beyond reach because of the many intricate and complex features of languages like Java. In this paper we present a transformational approach to the formal justification of proof systems for object-oriented programming languages. We focus on the following main characteristics of objects:

- objects possess (and *encapsulate*) their own (so-called instance) variables, and
- objects interact via *method* calls.

* Corresponding author at: Centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands.
E-mail addresses: apt@cwi.nl (K.R. Apt), F.S.de.Boer@cwi.nl (F.S. de Boer), olderog@informatik.uni-oldenburg.de (E.-R. Olderog).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2011.08.002

The execution of a method call involves a temporary *transfer* of control from the local state of the caller object to that of the called object (also referred to by *callee*). Upon termination of the method call the control returns to the local state of the caller. The method calls are the *only way* to transfer control from one object to another. We illustrate our approach by a syntax-directed transformation of the considered object-oriented programs to *recursive programs*. This transformation naturally suggests the corresponding proof rules. The main result of this paper is that the transformation preserves (relative) completeness.

To make this approach work a number of subtleties need to be taken care of. To start with, the 'base' language needs to be appropriately chosen. More precisely, to properly deal with the problem of avoiding methods calls on the **null** object we need a failure statement. In turn, to deal in a simple way with the call-by-value parameter mechanism we use parallel assignment and block statement. Further, to take care of the local variables of objects at the level of assertions we need to appropriately define the assertion language and deal with the substitution and aliasing.

We introduced this approach to the verification of object-oriented programs in our recent book [3] where we proved soundness. The aim of this paper is to provide a systematic and self-contained presentation which focuses on (relative) completeness and to explain how to extend this approach to other features of object-oriented programming. Readers interested in example correctness proofs may consult [3, pp. 226–237].

1.2. Related work

The origins of the proof theory for recursive method calls presented here can be traced back to [12]. However, in [12] the transformational approach to soundness and relational completeness was absent and failures were not dealt with. In [25] an extension to the typical object-oriented features of *inheritance* and *subtyping* is described. There is a large literature on assertional proof methods for object-oriented languages, notably for Java. For example, [17] discusses a weakest precondition calculus for Java programs with annotations in the Java Modeling Language (JML). JML can be used to specify Java classes and interfaces by adding annotations to Java source files. An overview of its tools and applications is provided in [8]. In [16] a Hoare logic for Java with abnormal termination caused by failures is described. However, this logic involves a major extension of the traditional Hoare logic to deal with failures for which the transformational approach breaks down.

Object-oriented programs in general give rise to dynamically evolving *pointer* structures as they occur in programming languages like Pascal. This leads to the problem of *aliasing*. There is a large literature on logics dealing with aliasing. One of the early approaches, focusing on the linked data structures, is described in [21]. A more recent approach is that of *separation logic* described in [28]. In [1] a Hoare logic for object-oriented programs is introduced based on an explicit representation of the global store in the assertion language. In [5] restrictions on aliasing are introduced to ensure encapsulation of classes in an object-oriented programming language with pointers and subtyping.

Recent work on assertional methods for object-oriented programming languages (see for example [6]) focuses on *object invariants* and a corresponding methodology for *modular* verification. In [22] also a class of invariants is introduced which support modular reasoning about complex object structures.

Formal justification of proof systems for object-oriented programming languages have been restricted to soundness (see for example [30] and [18]). Because of the many intricate and complex features of current object-oriented programming languages (relative) completeness remained largely beyond reach. Interestingly, in the above-mentioned [1] the use of the global store model is identified as a potential source of *incompleteness*.

1.3. Technical contributions

The proof system for object-oriented programs presented in our paper is based on an assertion language comparable to JML. This allows for the specification of dynamically evolving object structures at an abstraction level which coincides with that of the programming language: in this paper the only operations on objects we allow are testing for equality and dereferencing. Our transformation of the considered object-oriented programs to recursive programs preserves this abstraction level. As a consequence we have to adapt existing completeness proofs to recursive programs that use variables ranging over *abstract data types*, e.g., the type of objects.

In this paper we focus on *strong* partial correctness which requires absence of failures. Note that absence of failures is naturally expressed by a corresponding condition on the *initial* state, that is, by a corresponding notion of *weakest precondition*. Similarly, total correctness of recursive programs is also naturally expressed by weakest preconditions, see [2].

To express weakest preconditions over abstract data types in an assertion language [29] use a coding technique that requires a weak second-order language. In contrast, we introduce here a new state-based coding technique that allows us to express weakest preconditions over abstract data types in the presence of infinite arrays in a *first-order* assertion language.

Further, we generalize the original completeness proof of [13] for the partial correctness of recursive programs to weakest preconditions in order to deal with strong partial correctness. The completeness proof of [13] is based on the expression of the *graph* of a procedure call in terms of its strongest postcondition of a precondition which "freezes" the initial state by some fresh variables. If we use instead weakest preconditions to express the graph of a procedure call these freeze variables are used to denote the *final* state. Because of possible *divergence* or *failures* however we cannot eliminate in the precondition these freeze variables by existential quantification. As such the completeness proof of [13] breaks down. We show in this

paper how to restore completeness by the introduction of weakest preconditions which explicitly model divergence and failures.

1.4. Plan of the paper

In the next section we introduce a kernel language that consists of **while** programs augmented with the parallel assignment, the failure statement and the block statement, and describe its operational semantics. In Section 3 we extend this kernel language to a small object-oriented language that forms the subject of our considerations. In Section 4 we define an operational semantics of this language.

Then, in Section 5 we introduce a class of recursive programs, define a transformation of the object-oriented programs to recursive programs, and prove correctness of this transformation in an appropriate sense.

Next, in Section 6 we introduce the assertion language for object-oriented programs and extend the substitution operation to instance variables. In Section 7 we introduce the proof system that allows us to prove correctness of the considered object-oriented programs. Subsequently, in Section 8 we explain how soundness and relative completeness of this system can be established by reducing it to the analysis of a corresponding proof system for recursive procedures.

In Section 9 we prove relative completeness of our proof system for object-oriented programs on the basis of the transformation, addressing the issues described above. Finally, in Section 10 explain how this approach can be extended to deal with other features of object-oriented programming, like classes, inheritance and subtyping, and with total correctness. In Appendices A–C we list the rules defining the semantics of the kernel language, the introduced proof rules and the introduced proof systems.

2. Preliminaries

2.1. A kernel language

We assume at least two *basic* types, **integer** and **Boolean**, and for each $n \geqslant 1$ the *higher* types $T_1 \times \cdots \times T_n \to T$, where $T_1, \ldots, T_n, T$ are basic types. $T_1, \ldots, T_n$ are called *argument* types and T the *value* type. *Simple* variables are of a basic type and *array* variables of a higher type. By *Var* we denote the set of variable declarations. Usually, we omit the typing information and identify a variable declaration with the variable name. Out of typed variables and typed constants typed *expressions* are constructed. To deal with aliasing we use *conditional* expressions of the form **if** B **then** t_1 **else** t_2 **fi**. A *subscripted variable* is an expression $a[t_1, \ldots, t_n]$ for a suitably typed array variable a.

In this section we introduce the following small kernel programming language:

$$S ::= \mathit{skip} \mid u := t \mid \bar{x} := \bar{t} \mid S_1;\ S_2 \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{if } B \to S_1 \textbf{ fi} \mid$$
$$\textbf{while } B \textbf{ do } S_1 \textbf{ od} \mid \textbf{begin local } \bar{x} := \bar{t};\ S_1 \textbf{ end}$$

where S stands for a typical statement or program, u for a simple or subscripted variable, t for an expression (of the same type as u), and B for a Boolean expression. Further, $\bar{x} := \bar{t}$ is a parallel assignment, with $\bar{x} = x_1, \ldots, x_n$ a non-empty sequence of distinct simple variables and $\bar{t} = t_1, \ldots, t_n$ a sequence of expressions of the corresponding types. The parallel assignment plays a crucial role in our modeling of the parameter passing. The *failure statement* **if** $B \to S_1$ **fi** is used to check the condition B during the execution. It raises a failure if B is not satisfied. Thus it differs from the abbreviation **if** B **then** S **fi** $\equiv$ **if** B **then** S **else** *skip* **fi**. To distinguish between local and global variables, we use a *block statement* **begin local** $\bar{x} := \bar{t}; S_1$ **end**, where $\bar{x}$ is a non-empty sequence of simple distinct *local* variables, all of which are explicitly initialized by means of a parallel assignment $\bar{x} := \bar{t}$. We assume that the sets of local and global variables are disjoint.

For an expression t, we denote by $\mathit{var}(t)$ the set of all simple and array variables in t. Analogously, for a program S, we denote by $\mathit{var}(S)$ the set of all simple and array variables in S, and by $\mathit{change}(S)$ the set of all global simple and array variables that can be modified by S, i.e., the set of variables that appear on the left-hand side of an assignment in S outside of a subscript position of a subscripted variable.

2.2. ... and its semantics

We define the operational semantics of the kernel language in a standard way, using a structural operational semantics in the sense of Plotkin [27]. A *configuration* C is a pair $\langle S, \sigma \rangle$ consisting of a statement S that is to be executed and a *state* σ, i.e., a mapping that assigns to each variable (including local variables) of type T a value drawn from the set $\mathcal{D}_T$ denoted by type T.

Given a state σ and an expression t, we define in a standard way its semantics $\sigma(t)$, the value assigned to it by σ. Further, given a sequence of expressions $\bar{t}$ (in particular, a sequence of variables $\bar{x}$), we denote by $\sigma(\bar{t})$ the corresponding sequence of values assigned to $\bar{t}$ by σ.

We denote the set of states by Σ. Unless stated otherwise, the letters σ, τ range over Σ. We use a *special* state **fail** to represent an abnormal situation in a program execution, a *failure* in an execution of a program. We stipulate that **fail** $\notin \Sigma$. Sometimes to avoid confusion we refer to the elements of Σ as *proper* states.

We use the notion of a *state update* of a proper state σ, written as $\sigma[u := d]$, where u is a simple or subscripted variable of type T and d is an element of type T. If u is a simple variable then $\sigma[u := d]$ is the state that agrees with σ except for u where its value is d. If u is a subscripted variable, say $u \equiv a[t_1, \ldots, t_n]$, then $\sigma[u := d]$ is the state that agrees with σ except for the variable a where the value $\sigma(a)(\sigma(t_1), \ldots, \sigma(t_n))$ is changed to d. For the special state we define the update by $\mathbf{fail}[u := d] = \mathbf{fail}$. Further, the parallel update $\sigma[u_1, \ldots, u_n := d_1, \ldots, d_n]$ of distinct simple variables is the state that agrees with σ except for u_i where its value is d_i, for $i \in \{1, \ldots, n\}$.

A *transition* is a step $C \to C'$ between configurations. To express termination we use the empty statement E; a configuration $\langle E, \sigma\rangle$ denotes termination in the state σ. Transitions are specified by *transition axioms* and *rules*. The only transition axioms that are somewhat non-standard deal with the block statement and the failure statement. We write here $\sigma \models B$ to denote that B is true in the state σ.

- $\langle \mathbf{if}\ B \to S\ \mathbf{fi}, \sigma\rangle \to \langle S, \sigma\rangle$, where $\sigma \models B$,
- $\langle \mathbf{if}\ B \to S\ \mathbf{fi}, \sigma\rangle \to \langle E, \mathbf{fail}\rangle$, where $\sigma \models \neg B$,
- $\langle \mathbf{begin\ local}\ \bar{x} := \bar{t};\ S\ \mathbf{end}, \sigma\rangle \to \langle \bar{x} := \bar{t};\ S;\ \bar{x} := \sigma(\bar{x}), \sigma\rangle$.

The last axiom ensures that the local variables are initialized as prescribed by the parallel assignment and that upon termination the local variables are restored to their initial values, held at the beginning of the block statement. This way we implicitly model a *stack discipline* for, possibly nested, blocks. The other transition axioms and rules are standard (see Appendix A).

The *partial correctness semantics* is a mapping $\mathcal{M}[\![S]\!] : \Sigma \to \mathcal{P}(\Sigma)$ defined by

$$\mathcal{M}[\![S]\!](\sigma) = \big\{\tau \in \Sigma \mid \langle S, \sigma\rangle \to^* \langle E, \tau\rangle\big\},$$

where $\to^*$ denotes the reflexive, transitive closure of $\to$. The *strong partial correctness semantics* is a mapping $\mathcal{M}_{sp}[\![S]\!] : \Sigma \to \mathcal{P}(\Sigma \cup \{\mathbf{fail}\})$ defined by

$$\mathcal{M}_{sp}[\![S]\!](\sigma) = \big\{\tau \in \Sigma \cup \{\mathbf{fail}\} \mid \langle S, \sigma\rangle \to^* \langle E, \tau\rangle\big\}.$$

So for all S and σ we have $\mathbf{fail} \notin \mathcal{M}[\![S]\!](\sigma)$, while for some S and σ we can have $\mathbf{fail} \in \mathcal{M}_{sp}[\![S]\!](\sigma)$. In the latter case we say that S *can fail* when started in σ. We extend these semantic mappings to *sets* of states, $X \subseteq \Sigma$, by collecting all results obtained for the individual states $\sigma \in X$.

3. Object-oriented programs: syntax

To define the syntax of the considered object-oriented programming language we introduce a new basic type **object** which denotes an infinite set of objects $\mathcal{D}_{\mathbf{object}}$.

3.1. Expressions

An expression of type **object** denotes an object. Simple variables of type **object** and array variables with value type **object** are called *object* variables. We distinguish the simple object variable **this** which in each state denotes the currently executing object.

Besides the set *Var* of variable declarations defined in Section 2 we now introduce a new set *IVar* of instance variable declarations (so $Var \cap IVar = \emptyset$). An *instance variable* can be a simple variable or an array variable. Thus we now have two kinds of variable declarations: the up till now considered declarations of *normal variables* (*Var*), which are shared, and the new declarations of instance variables (*IVar*), which are owned by objects. As before we identify each variable declaration with the variable name. Out of instance array variables we construct, as in the case of normal array variables, *subscripted instance variables*.

For simplicity we assume that each object owns the same set of instance variables. Each object has its own local state which assigns values to the instance variables. We stipulate that **this** is a normal variable, that is, $\mathbf{this} \in Var$.

The only operation of a higher type which involves the basic type **object** (as argument type or as value type) is the equality $=_{\mathbf{object}}$ (abbreviated by $=$). Finally, we use the constant **null** of type **object** to represent the *void reference*, a special construct which does not have a local state.

Summarizing, the set of expressions defined in Section 2 is extended by the introduction of the basic type **object**, the constant **null** of type **object**, and the set *IVar* of (simple and array) instance variables. Object expressions, i.e., expressions of type **object**, can only be compared for equality. A variable is either a normal variable (in *Var*) or an instance variable (in *IVar*). Simple variables (in $Var \cup IVar$) can now be of type **object**. Also the argument and the value types of array variables (in $Var \cup IVar$) can be of type **object**. Finally, we have the distinguished normal object variable **this**.

3.2. Programs

For object-oriented programs we extend the syntax of the kernel language introduced in Section 2. Assignments to instance variables are introduced as follows:

$$S ::= u := t,$$

where $u \in IVar$ is a simple or subscripted (instance) variable. Method calls are described by the clause

$$S ::= s.m(t_1, \ldots, t_n),$$

where $n \geqslant 0$. Here the object expression s denotes the *called object*, the identifier m denotes a method and $t_1, \ldots, t_n$ are the *actual parameters*, which are expressions of a basic type. A method is defined by means of a *declaration*

$$m(u_1, \ldots, u_n) :: S,$$

where the formal parameters $u_1, \ldots, u_n \in Var$ are of a basic type and S is a statement called the *method body*. Since the statements now include method calls, we allow for mutually *recursive* methods. However, the declarations cannot be nested, so we do not allow for nested methods.

The instance variables appearing in the body S of a method declaration are owned by the executing object, which is denoted by the variable **this**. To ensure correct use of the variable **this** we disallow assignments to the variable **this**. However, when describing the semantics of method calls, we do use 'auxiliary' block statements in which the variable **this** *is* used as a local variable, so in particular, it is initialized (and hence modified). Further, to ensure that instance variables are permanent, we require that in each block statement instance variables are not used as local variables.

Apart from denoting the callee of a method call, object expressions can appear in Boolean expressions. Further, we allow for assignments to object variables.

An *object-oriented program* consists of a *main statement* S built according to the syntax of this section and a given set D of method declarations such that each method used has a unique declaration in D and each method call refers to a method declared in D. We assume that method calls are *well-typed*, i.e., the numbers of formal and actual parameters agree and for each parameter position the types of the corresponding actual and formal parameters coincide. As before, name clashes between local variables and global variables are resolved by assuming that no local variable of S or D occurs freely (i.e., as a global variable) in S or D. If D is clear from the context we refer to the main statement as an object-oriented program.

Example 3.1. Consider the object-oriented program

$$S \equiv \textbf{this}.find(z)$$

in the context of the recursive method declaration

$$find(u) :: \textbf{if } u \neq \textbf{this then } next.find(u) \textbf{ fi}.$$

We assume that the actual parameter z, the formal parameter u, and the instance variable *next* are of type **object**. The idea is that S checks whether a list of objects linked via the pointer *next* contains an object stored in the actual parameter z. The search through the list starts at the object stored in the variable **this**.

4. Object-oriented programs: semantics

In this section we define the semantics of the introduced object-oriented programs. We first define the semantics of expressions. It requires an extension of the definition of state. Subsequently we introduce a revised definition of an update of a state and provide transition axioms concerned with the newly introduced programming constructs.

4.1. Semantics of expressions

The main difficulty in defining the semantics of expressions is of course how to deal properly with the instance variables. Each instance variable has a different version ('instance') in each object. Conceptually, when defining the semantics of an instance variable u we view it as a variable of the form **this**.u, where **this** represents the current object. So, given a proper state σ and a simple instance variable x we first determine the current object o, which is $\sigma(\textbf{this})$. Then we determine the *local state* of this object, which is $\sigma(o)$, or $\sigma(\sigma(\textbf{this}))$, and subsequently apply this local state to the considered instance variable x. This means that given a proper state σ the value assigned to the instance variable x is $\sigma(o)(x)$, or, written out in full, $\sigma(\sigma(\textbf{this}))(x)$. This two-step procedure is at the heart of the definition of semantics of an expression given below.

Next, we introduce a value $\textbf{null} \in \mathcal{D}_{\textbf{object}}$. So in each proper state each variable of type **object** equals some object of $\mathcal{D}_{\textbf{object}}$, which can be the **null** object. A proper state σ now additionally assigns to each object $o \in \mathcal{D}_{\textbf{object}}$ its local state $\sigma(o)$. In turn, a local state $\sigma(o)$ of an object o assigns a value of appropriate type to each instance variable. Note that by definition a proper state also assigns to **null** a local state. However, by Lemma 4.2 from Section 4.3 below this state is not reachable in any computation.

Note that the local state of the current object $\sigma(\textbf{this})$ is given by $\sigma(\sigma(\textbf{this}))$. Further, note that in particular, if an instance variable x is of type **object**, then for each object $o \in \mathcal{D}_{\textbf{object}}$, $\sigma(o)(x)$ is either **null** or an object $o' \in \mathcal{D}_{\textbf{object}}$, whose local state is $\sigma(o')$, i.e., $\sigma(\sigma(o)(x))$. This application of σ can of course be nested, to get local states of the form $\sigma(\sigma(\sigma(o)(x))(x))$, etc.

To illustrate the notion of a state consider Fig. 1. The current object is represented by a pointer to its memory region. Each occurrence of the variable x is here an instance variable of a different object. In contrast, the normal variables, in particular **this**, form the global component of the state.

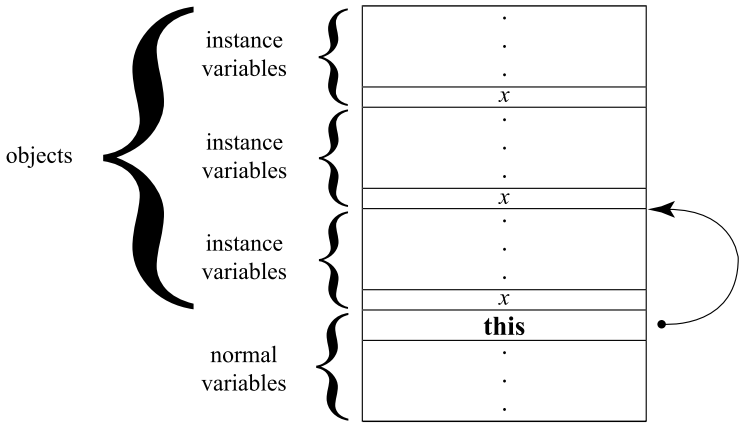

Fig. 1. A state.

We need to extend the semantics $\sigma(s)$ of an expression s in a proper state σ (cf. Section 2.2) by the following clauses:

- if $s \equiv \textbf{null}$ then $\sigma(s) = \textbf{null}$, so the meaning of the constant **null** (representing the void reference) is the **null** object,
- if $s \equiv x$ for some simple instance variable x then $\sigma(s) = \sigma(o)(x)$, where $o = \sigma(\textbf{this})$, so in expanded form this is

$$\sigma(x) = \sigma\big(\sigma(\textbf{this})\big)(x), \tag{1}$$

- if $s \equiv a[s_1, \ldots, s_n]$ for some instance array variable a then

$$\sigma(s) = \sigma(o)(a)\big(\sigma(s_1), \ldots, \sigma(s_n)\big),$$

where $o = \sigma(\textbf{this})$.

4.2. Updates of states

Next, we revise the definition of a state update for the case of instance variables. Consider a proper state σ, a simple instance variable x, and a value d belonging to the type of x. To perform the corresponding *state update* of σ on x to d, written as $\sigma[x := d]$, we first identify the current object o, which is $\sigma(\textbf{this})$ and its local state, which is $\sigma(o)$, or $\sigma(\sigma(\textbf{this}))$, that we denote by τ. Then we perform the appropriate update on the state τ. So the desired update of σ is achieved by modifying τ to $\tau[x := d]$.

In general, let u be a (possibly subscripted) instance variable of type T and τ a local state. We define for $d \in \mathcal{D}_T$

$$\tau[u := d]$$

analogously to the standard definition of state update for normal variables. Furthermore, we define for an object $o \in \mathcal{D}_{\textbf{object}}$ and local state τ, the state update $\sigma[o := \tau]$ by

$$\sigma[o := \tau]\big(o'\big) = \begin{cases} \tau & \text{if } o = o', \\ \sigma(o') & \text{otherwise.} \end{cases}$$

We are now in a position to define the state update $\sigma[u := d]$ for a (possibly subscripted) instance variable u of type T and $d \in \mathcal{D}_T$, as follows:

$$\sigma[u := d] = \sigma\big[o := \tau[u := d]\big],$$

where $o = \sigma(\textbf{this})$ and $\tau = \sigma(o)$. Note that the state update $\sigma[o := \tau[u := d]]$ assigns to the current object o the update $\tau[u := d]$ of its local state τ. In its fully expanded form we get the following difficult to parse definition of a state update:

$$\sigma[u := d] = \sigma\big[\sigma(\textbf{this}) := \sigma\big(\sigma(\textbf{this})\big)[u := d]\big].$$

Example 4.1. Let x be a Boolean instance variable, $o = \sigma(\textbf{this})$, and $\tau = \sigma(o)$. Then

$$\begin{aligned} & \sigma[x := \textbf{true}](x) \\ = \quad & \big\{(1) \text{ with } \sigma \text{ replaced by } \sigma[x := \textbf{true}]\big\} \\ & \sigma[x := \textbf{true}]\big(\sigma[x := \textbf{true}](\textbf{this})\big)(x) \\ = \quad & \big\{\text{by the definition of state update, } \sigma[x := \textbf{true}](\textbf{this}) = \sigma(\textbf{this}) = o\big\} \end{aligned}$$

$$
\begin{aligned}
& \sigma[x := \mathbf{true}](o)(x) \\
= \quad & \{\text{definition of state update } \sigma[x := \mathbf{true}]\} \\
& \sigma\big[o := \tau[x := \mathbf{true}]\big](o)(x) \\
= \quad & \{\text{definition of state update } \sigma\big[o := \tau[x := \mathbf{true}]\big]\} \\
& \tau[x := \mathbf{true}](x) \\
= \quad & \{\text{definition of state update } \tau[x := \mathbf{true}]\} \\
& \mathbf{true}.
\end{aligned}
$$

4.3. Semantics of programs

For the operational semantics of the considered programs we introduce two transition axioms that deal with assignments to simple or subscripted instance variables u and with method calls $s.m(\bar{t})$, where $\bar{t}$ is the list of actual parameters.

- $\langle u := t, \sigma\rangle \to \langle E, \sigma[u := \sigma(t)]\rangle$,
- $\langle s.m(\bar{t}), \sigma\rangle \to \langle \mathbf{if}\ s \neq \mathbf{null} \to \mathbf{begin\ local\ this}, \bar{u} := s, \bar{t};\ \ S\ \mathbf{end\ fi}, \sigma\rangle$, where $m(\bar{u}) :: S \in D$.

This clarifies that we use the stack discipline to handle the method calls. Indeed, the method body S is executed in the state in which the current object (denoted by the variable **this**) becomes $\sigma(s)$, and upon termination of the method body S the current object is restored to its previous value $\sigma(\mathbf{this})$ using the parallel assignment $\sigma[\mathbf{this}, \bar{u} := \sigma(s, \bar{t})]$. The use of the failure statement implies that if in the considered state σ the called object s equals the void reference (it equals **null**), then the method call yields a failure.

Lemma 4.2 *(Safety). For every statement S that can arise during an execution of an object-oriented program and every proper state σ, the following hold.*

(i) ***Absence of* null *Reference*:** *if $\sigma(\mathbf{this}) \neq \mathbf{null}$ and $\langle S, \sigma\rangle \to \langle S_1, \tau\rangle$, then $\tau(\mathbf{this}) \neq \mathbf{null}$.*
(ii) ***Type Safety*:** *if S is well-typed and $\langle S, \sigma\rangle \to \langle S_1, \tau\rangle$ holds, then also S_1 is well-typed.*

Proof. (i) If $S \not\equiv E$ then any configuration $\langle S, \sigma\rangle$ has a successor in the transition relation $\to$. To prove the preservation of the assumed property of the state it suffices to consider the execution of an assignment $\mathbf{this} := s$. Each such assignment arises only within the context of the block statement in the corresponding transition axiom and is activated in a state σ such that $\sigma(s) \neq \mathbf{null}$. This yields a state τ such that $\tau(\mathbf{this}) \neq \mathbf{null}$.

(ii) Except for method calls, the statements on the right-hand side of the transition axioms are composed of the substatements of the statement on the left-hand side of the transition axiom, which are well-typed by assumption. Further, by the second transition axiom above, well-typed method calls lead to well-typed parallel assignments in the block statements. □

When considering verification of object-oriented programs we shall only consider computations that start in a proper state σ such that $\sigma(\mathbf{this}) \neq \mathbf{null}$, i.e., in a state in which the current object differs from the void reference. The Safety Lemma 4.2 implies that such computations never lead to a proper state in which this inequality is violated.

The *partial correctness semantics* $\mathcal{M}[\![S]\!]$ and the *strong partial correctness semantics* $\mathcal{M}_{sp}[\![S]\!]$ of object-oriented programs S are defined as for the kernel language.

5. Transformation to recursive programs

In this section we show that object-oriented programs introduced in the previous section can be translated by means of a simple syntax-driven transformation to recursive programs with parameters. Intuitively, for each method the current object is made into an explicit parameter of the corresponding recursive procedure.

5.1. Recursive programs

As a preparation we introduce recursive programs by adding recursive procedures with call-by-value parameters to the kernel language. *Procedure calls* with parameters are introduced by the grammar rule

$$S ::= P(t_1, \ldots, t_n),$$

where P is a procedure identifier and $t_1, \ldots, t_n$, with $n \geqslant 0$, are expressions called *actual parameters*. Procedures are defined by *declarations* of the form

$$P(u_1, \ldots, u_n) :: S,$$

where $u_1, \ldots, u_n$ are distinct simple variables, called *formal parameters* of the procedure P and S is the *body* of the procedure P.

We assume a given set of procedure declarations D such that each procedure that appears in D has a unique declaration in D. A *recursive program* consists of a *main statement* S built according to the syntax of this section and a given set D of procedure declarations such that all procedures whose calls appear in the considered recursive programs are declared in D. So we allow mutually recursive procedures but not nested procedures. We assume that procedure calls are *well-typed* in the same sense as method calls. As in the case of the object-oriented programs, name clashes between local variables and global variables are resolved by assuming that no local variable of S or D occurs freely in S or D.

5.1.1. Semantics

For recursive programs we extend the operational semantics of the kernel language by the following transition axiom that describes the *call-by-value* parameter mechanism.

$$\big\langle P(\bar{t}), \sigma \big\rangle \to \langle \textbf{begin local } \bar{u} := \bar{t}; S \textbf{ end}, \sigma \rangle, \quad \text{where } P(\bar{u}) :: S \in D.$$

This yields for a recursive program S the semantics $\mathcal{M}[\![S]\!]$ and $\mathcal{M}_{sp}[\![S]\!]$.

Note that thanks to the semantics of the block statement this axiom correctly handles the clash between formal and actual parameters. For example for $P(u) :: S \in D$ we get, as desired,

$$\big\langle P(u+1), \sigma \big\rangle \to^* \big\langle S; u := \sigma(u), \sigma\big[u := \sigma(u+1)\big] \big\rangle.$$

5.2. Transformation

We now define a formal relation between object-oriented programs and recursive programs. We assume the class of recursive programs that use normal variables whose type may involve the basic type **object** and the class of object-oriented programs, as defined in Section 3. Further, we assume for every declaration of an instance variable u of a basic type T a declaration in *Var* of a normal array variable u of type

$$\textbf{object} \to T.$$

Similarly, we assume for every declaration of an instance variable a of a higher type $T_1 \times \cdots \times T_n \to T$ a declaration in *Var* of a normal array variable a of type

$$\textbf{object} \times T_1 \times \cdots \times T_n \to T.$$

A normal array variable of type

$$\textbf{object} \to T$$

in the recursive program will represent the instance variable of basic type T in the corresponding object-oriented program, and a normal array variable of type

$$\textbf{object} \times T_1 \times \cdots \times T_n \to T$$

in the recursive program will represent an instance variable of the corresponding object-oriented program of type $T_1 \times \cdots \times T_n \to T$.

Given an 'object-oriented' state σ we denote by $\Theta(\sigma)$ the 'normal' state which represents the instance variables as normal variables. On normal variables of type T the states σ and $\Theta(\sigma)$ agree and are of type $Var \to \mathcal{D}_T$. For instance variables of basic type T the state σ is of type

$$\mathcal{D}_{\textbf{object}} \to (IVar \to \mathcal{D}_T),$$

the corresponding state $\Theta(\sigma)$ is of type

$$Var \to (\mathcal{D}_{\textbf{object}} \to \mathcal{D}_T),$$

and for instance array variables of type $T_1 \times \cdots \times T_n \to T$ the state σ is of type

$$\mathcal{D}_{\textbf{object}} \to \big(IVar \to (\mathcal{D}_{T_1} \times \cdots \times \mathcal{D}_{T_n} \to \mathcal{D}_T)\big),$$

and the corresponding state $\Theta(\sigma)$ is of type

$$Var \to (\mathcal{D}_{\textbf{object}} \times \mathcal{D}_{T_1} \times \cdots \times \mathcal{D}_{T_n} \to \mathcal{D}_T).$$

Formally, $\Theta(\sigma)$ is defined as follows:

- $\Theta(\textbf{fail}) = \textbf{fail}$,
- $\Theta(\sigma)(x) = \sigma(x)$, for every normal variable x,
- $\Theta(\sigma)(z)(o) = \sigma(o)(z)$, for every object $o \in \mathcal{D}_{\textbf{object}}$ and normal array variable z of type $\textbf{object} \to T$ on the left-hand side of the equation corresponding to an instance variable z of a basic type T on the right-hand side of the equation,

- $\Theta(\sigma)(a)(o, d_1, \ldots, d_n) = \sigma(o)(a)(d_1, \ldots, d_n)$, for every object $o \in \mathcal{D}_{\mathbf{object}}$ and normal array variable a of type $\mathbf{object} \times T_1 \times \cdots \times T_n \to T$ on the left-hand side of the equation corresponding to an instance array variable a of type $T_1 \times \cdots \times T_n \to T$ on the right-hand side of the equation, and $d_i \in \mathcal{D}_{T_i}$, for $i \in \{1, \ldots, n\}$.

Next, we define for every expression s of the object-oriented programming language the 'normal' expression $\Theta(s)$ of the recursive program by induction on the structure of s, with the following base cases:

- $\Theta(x) \equiv x$, for every normal variable x,
- $\Theta(x) \equiv x[\mathbf{this}]$, for every instance variable x of a basic type,
- $\Theta(a[s_1, \ldots, s_n]) \equiv a[\mathbf{this}, \Theta(s_1), \ldots, \Theta(s_n)]$, for every instance array variable a.

The first case in particular yields $\Theta(\mathbf{this}) \equiv \mathbf{this}$. The following lemma clarifies the outcome of this transformation.

Lemma 5.1 *(Translation). For all proper states σ the following hold.*

(i) *For all expressions s,*

$$\sigma(s) = \Theta(\sigma)\big(\Theta(s)\big),$$

where $\Theta(\sigma)(\Theta(s))$ refers to the standard semantics of expressions which involve only normal variables.

(ii) *For all (possibly subscripted) instance variables u and values d of the same type as u,*

$$\Theta\big(\sigma[u := d]\big) = \Theta(\sigma)\big[\Theta(u) := d\big].$$

Proof. By straightforward induction on the structure of s and case analysis on the structure of u. □

Next, we extend by structural induction the transformation Θ to statements of the considered object-oriented language. The failure statement is used to take care of the method calls on the void reference. We prove then that this transformation preserves both partial and strong partial correctness semantics.

- $\Theta(skip) \equiv skip$,
- $\Theta(\bar{x} := \bar{t}) \equiv \bar{x} := \Theta(\bar{t})$,
- $\Theta(u := s) \equiv \Theta(u) := \Theta(s)$,
- $\Theta(s.m(s_1, \ldots, s_n)) \equiv \mathbf{if}\ \Theta(s) \neq \mathbf{null} \to m(\Theta(s), \Theta(s_1), \ldots, \Theta(s_n))\ \mathbf{fi}$,
- $\Theta(S_1;\ S_2) \equiv \Theta(S_1);\ \Theta(S_2)$,
- $\Theta(\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}) \equiv \mathbf{if}\ \Theta(B)\ \mathbf{then}\ \Theta(S_1)\ \mathbf{else}\ \Theta(S_2)\ \mathbf{fi}$,
- $\Theta(\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}) \equiv \mathbf{while}\ \Theta(B)\ \mathbf{do}\ \Theta(S)\ \mathbf{od}$,
- $\Theta(\mathbf{if}\ B \to S\ \mathbf{fi}) \equiv \mathbf{if}\ \Theta(B) \to \Theta(S)\ \mathbf{fi}$,
- $\Theta(\mathbf{begin\ local}\ \bar{u} := \bar{t}; S\ \mathbf{end}) \equiv \mathbf{begin\ local}\ \bar{u} := \Theta(\bar{t}); \Theta(S)\ \mathbf{end}$, where $\Theta(\bar{t})$ denotes the result of applying Θ to the sequence of expressions $\bar{t}$.

So the translation of a method call $s.m(s_1, \ldots, s_n)$ transforms the called object s into an additional actual parameter of a call of the procedure m. Additionally a check for a failure is added. Finally, we transform every method declaration

$$m(u_1, \ldots, u_n) :: S$$

into a procedure declaration

$$m(\mathbf{this}, u_1, \ldots, u_n) :: \Theta(S).$$

So the distinguished normal variable **this** is added as an additional formal parameter of the procedure m. This way the set D of method declarations is transformed into the set

$$\Theta(D) = \big\{m(\mathbf{this}, u_1, \ldots, u_n) :: \Theta(S) \;\big|\; m(u_1, \ldots, u_n) :: S \in D\big\}$$

of the corresponding procedure declarations.

Example 5.2. Consider the object-oriented program

$$S \equiv y.add(1);\ y.add(2),$$

where y is a normal variable of type **object**, in the context of the declaration

$$D = \big\{add(x) :: sum := sum + x\big\},$$

where the formal parameter x is of type **integer** and *sum* is an instance variable, also of type **integer**. Then the transformation Θ yields

$$\Theta(S) \equiv \textbf{if } y \neq \textbf{null} \to add(y,1) \textbf{ fi};\ \textbf{if } y \neq \textbf{null} \to add(y,2) \textbf{ fi}$$

and

$$\Theta(D) = \{add(\textbf{this}, x) :: sum[\textbf{this}] := sum[\textbf{this}] + x\}$$

as the corresponding recursive program.

5.3. Correctness proof

We have the following crucial correspondence between an object-oriented program S and its transformation $\Theta(S)$.

Lemma 5.3 *(Transformation). For all well-typed object-oriented programs S, all sets of method declarations D, all proper states σ, and all proper or* **fail** *states τ,*

$$\langle S, \sigma\rangle \to^* \langle E, \tau\rangle \quad \textit{iff} \quad \big\langle \Theta(S), \Theta(\sigma)\big\rangle \to^* \big\langle E, \Theta(\tau)\big\rangle.$$

Proof. We prove only the ($\Rightarrow$) direction. We proceed by induction on the number of the axiom and rule applications used in the computation $\langle S, \sigma\rangle \to^* \langle E, \tau\rangle$.

The only non-trivial case arises when S begins with a method call, that is, is of the form $s.m(\bar{t}); S_1$. By the assumption,

$$\big\langle s.m(\bar{t}); S_1, \sigma\big\rangle \to \langle \textbf{begin local this}, \bar{u} := s, \bar{t};\ S \textbf{ end}; S_1, \sigma\rangle \to^* \langle E, \tau\rangle,$$

where $\sigma(s) \neq \textbf{null}$ and $m(\bar{u}) :: S \in D$. So by the induction hypothesis and definition of Θ,

$$\big\langle \Theta(\textbf{begin local this}, \bar{u} := s, \bar{t};\ S \textbf{ end}); \Theta(S_1), \Theta(\sigma)\big\rangle \to^* \big\langle E, \Theta(\tau)\big\rangle.$$

Note that

$$\Theta\big(s.m(\bar{t}); S_1\big) \equiv \textbf{if } \Theta(s) \neq \textbf{null} \to m\big(\Theta(s), \Theta(\bar{t})\big) \textbf{ fi}; \Theta(S_1).$$

By the Translation Lemma 5.1(i), we have $\Theta(\sigma)(\Theta(s)) \neq \textbf{null}$. So by definition of the semantics of recursive programs and definition of Θ,

$$\begin{aligned}&\big\langle \textbf{if } \Theta(s) \neq \textbf{null} \to m\big(\Theta(s), \Theta(\bar{t})\big) \textbf{ fi}; \Theta(S_1), \Theta(\sigma)\big\rangle\\ &\quad \to^* \big\langle \Theta(\textbf{begin local this}, \bar{u} := s, \bar{t};\ S \textbf{ end}); \Theta(S_1), \Theta(\sigma)\big\rangle,\end{aligned}$$

which concludes the proof. □

Finally, the following theorem establishes the correctness of the transformation Θ as a homomorphism. We extend here Θ to a (possibly empty) set of states in an obvious way.

Theorem 5.4 *(Correctness of Θ). For all well-typed object-oriented programs S, all sets of method declarations D, and all proper states σ the following hold*:

(i) $\Theta(\mathcal{M}[\![S]\!](\sigma)) = \mathcal{M}[\![\Theta(S)]\!](\Theta(\sigma))$,
(ii) $\Theta(\mathcal{M}_{sp}[\![S]\!](\sigma)) = \mathcal{M}_{sp}[\![\Theta(S)]\!](\Theta(\sigma))$,

where S is considered in the context of the set D and the corresponding recursive program $\Theta(S)$ in the context of the set of procedure declarations $\Theta(D)$.

Proof. The claim is a direct consequence of the Transformation Lemma 5.3. □

6. Assertion language

6.1. Syntax and semantics

Expressions of the programming language only refer to the local state of the executing object and do not allow us to distinguish between different versions of the instance variables. In the assertions we need to be more explicit. So we introduce the set of *global expressions* which extends the set of expressions of the object-oriented programming language introduced in Section 3 by the following additional clauses:

- if s is a global expression of type **object** and x is an instance variable of a basic type T then $s.x$ is a global expression of type T,
- if s is a global expression of type **object**, $s_1, \ldots, s_n$ are global expressions of type $T_1, \ldots, T_n$, and a is an array instance variable of type $T_1 \times \cdots \times T_n \to T$ then $s.a[s_1, \ldots, s_n]$ is a global expression of type T.

In particular, every expression of the programming language is also a global expression.

Example 6.1. Consider a normal integer variable i, a normal variable x of type **object**, a normal array variable a of type **integer** $\to$ **object**, and an instance variable *next* of type **object**. Using them we can generate the following global expressions:

$$next,\ next.next,\ x.next,\ x.next.next,\ a[i].next,\ etc.,$$

all of type **object**. In contrast, $next.x$ is not a global expression, since x is not an instance variable.

We call a global expression of the form $s.u$ a *navigation expression* since it allows one to *navigate* through the local states of the objects. For example, the global expression $next.next$ refers to the object that can be reached by 'moving' to the object denoted by the value of *next* of the current object **this** and evaluating the value of its variable *next*.

We define the semantics of global expressions by extending the semantics of expressions given in Section 4.1 as follows:

- for a simple instance variable x of type T,

$$\sigma(s.x) = \sigma(o)(x),$$

  where $\sigma(s) = o$,
- for an instance array variable a with value type T,

$$\sigma\big(s.a[s_1, \ldots, s_n]\big) = \sigma(o)(a)\big(\sigma(s_1), \ldots, \sigma(s_n)\big),$$

  where $\sigma(s) = o$.

So for a simple or subscripted instance variable u the semantics of u and **this**.u coincide, that is, for all proper states σ we have $\sigma(u) = \sigma(\mathbf{this}.u)$. In other words, we can view an instance variable u as an abbreviation for the global expression **this**.u.

Note that this semantics also provides meaning to global expressions of the form **null**.u. However, such expressions are meaningless when specifying correctness of programs because the local state of the **null** object can never be reached in computations starting in a proper state σ such that $\sigma(\mathbf{this}) \neq \mathbf{null}$ (see the Safety Lemma 4.2).

Example 6.2. If x is an object variable and σ a proper state with $\sigma(x) \neq \mathbf{null}$, then for all simple instance variables y we have $\sigma(x.y) = \sigma(\sigma(x))(y)$.

Assertions are constructed from global Boolean expressions by adding quantification over simple *normal* variables. We use p, q as typical letters for assertions. For a state σ and an assertion p we write $\sigma \models p$ if σ *satisfies* p. Let $[\![p]\!]$ denote the set of proper states satisfying p, so $[\![p]\!] = \{\sigma \in \Sigma \mid \sigma \models p\}$. So $\sigma \models p$ iff $\sigma \in [\![p]\!]$.

6.2. Substitution and aliasing

We write $s[u := t]$ for the result of substituting an expression t for a simple or subscripted normal variable u in an expression s. We call $[u := t]$ a *substitution*. For a simple variable u this is defined in the customary way. Also it is straightforward how to define the simultaneous substitution $s[\bar{x} := \bar{t}]$ involving a sequence of simple variables.

However, for a subscripted variable u, the problem of *aliasing*, i.e., when syntactically different subscripted variables denote the same location, has to be taken care of. Following [11] we handle it using the conditional expressions. For example,

$$min\big(a[x], y\big)\big[a[1] := 2\big] \equiv \mathbf{if}\ x = 1\ \mathbf{then}\ min(2, y)\ \mathbf{else}\ min\big(a[x], y\big)\ \mathbf{fi}.$$

The conditional expression checks whether $a[x]$ and $a[1]$ are *aliases* of the same location. If so, the substitution of 2 for $a[1]$ results in $a[x]$ being replaced by 2, otherwise the substitution has no effect.

Intuitively, in a given state σ the substituted expression $s[u := t]$ describes the same value as the expression s evaluated in the updated state $\sigma[u := \sigma(t)]$, which arises after the assignment $u := t$ has been executed in σ. We shall later need the details of the definition of $s[u := t]$, so let us recall it here. It proceeds by induction on the structure of s. The cases dealing with subscripted variables are as follows:

- if $s \equiv a[s_1, \ldots, s_n]$ for some array a, and u is a simple variable or a subscripted variable $b[t_1, \ldots, t_m]$ with $a \not\equiv b$, then

$$s[u := t] \equiv a\big[s_1[u := t], \ldots, s_n[u := t]\big],$$

- if $s \equiv a[s_1, \ldots, s_n]$ for some array a and $u \equiv a[t_1, \ldots, t_n]$ then

$$s[u := t] \equiv \textbf{if} \bigwedge_{i=1}^{n} s'_i = t_i \textbf{ then } t \textbf{ else } a\big[s'_1, \ldots, s'_n\big] \textbf{ fi}$$

where $s'_i \equiv s_i[u := t]$ for $i \in \{1, \ldots, n\}$.

The most complicated case is the second clause for subscripted variables. Here the conditional expression

$$\textbf{if} \bigwedge_{i=1}^{n} s'_i = t_i \textbf{ then } \ldots \textbf{ else } \ldots \textbf{ fi}$$

checks whether, for any given proper state σ, the expression $s \equiv a[s_1, \ldots, s_n]$ in the updated state $\sigma[u := \sigma(t)]$ and the expression $u \equiv a[t_1, \ldots, t_n]$ in the state σ are aliases. For this check the substitution $[u := t]$ needs to applied inductively to all subscripts $s_1, \ldots, s_n$ of $a[s_1, \ldots, s_n]$. In case of an alias $s[u := t]$ yields t. Otherwise, the substitution is applied inductively to the subscripts $s_1, \ldots, s_n$ of $a[s_1, \ldots, s_n]$.

We now extend the definition of the outcome $s[u := t]$ of the substitution to the case of instance variables u and global expressions s and t constructed from them. Let u be a simple or subscripted instance variable and s and t global expressions. In general, the substitution $[u := t]$ replaces every possible alias $e.u$ of u by t. In addition to the possible aliases of subscripted variables, we now also have to consider the possibility that the global expression $e[u := t]$ denotes the current object **this**. This explains the use of conditional expressions below.

Here are the main cases of the definition of the substitution operation $s[u := t]$:

- if $s \equiv x \in Var$ then

$$s[u := t] \equiv s,$$

- if $s \equiv e.u$ and u is a simple instance variable then

$$s[u := t] \equiv \textbf{if } e' = \textbf{this then } t \textbf{ else } e'.u \textbf{ fi},$$

where $e' \equiv e[u := t]$,

- if $s \equiv e.a[s_1, \ldots, s_n]$ and $u \equiv a[t_1, \ldots, t_n]$ then

$$s[u := t] \equiv \textbf{if } e' = \textbf{this} \wedge \bigwedge_{i=1}^{n} s'_i = t_i \textbf{ then } t \textbf{ else } e'.a\big[s'_1, \ldots, s'_n\big] \textbf{ fi},$$

where $e' \equiv e[u := t]$ and $s'_i \equiv s_i[u := t]$ for $i \in \{1, \ldots, n\}$.

The following example should clarify this definition.

Example 6.3. Suppose that $s \equiv \textbf{this}.u$. Then

$$\begin{aligned} &\textbf{this}.u[u := t] \\ &\quad \equiv \textbf{if this}[u := t] = \textbf{this then } t \textbf{ else } \ldots \textbf{ fi} \\ &\quad \equiv \textbf{if this} = \textbf{this then } t \textbf{ else } \ldots \textbf{ fi}. \end{aligned}$$

So $\textbf{this}.u[u := t]$ and t are equal in the sense that for all proper states σ we have $\sigma(\textbf{this}.u[u := t]) = \sigma(t)$.

Next, suppose that $s \equiv \textbf{this}.a[x]$, where x is a simple variable. Then

$$\begin{aligned} &\textbf{this}.a[x]\big[a[x] := t\big] \\ &\quad \equiv \textbf{if this}\big[a[x] := t\big] = \textbf{this} \wedge x\big[a[x] := t\big] = x \textbf{ then } t \textbf{ else } \ldots \textbf{ fi} \\ &\quad \equiv \textbf{if this} = \textbf{this} \wedge x = x \textbf{ then } t \textbf{ else } \ldots \textbf{ fi}. \end{aligned}$$

So $\textbf{this}.a[x][a[x] := t]$ and t are equal.

The substitution operation is then extended to assertions by properly taking care of quantification. We have the following lemma that relates for instance variables the effect of substitution to the state update.

Lemma 6.4 *(Substitution of Instance Variables). For all global expressions s and t, all assertions p, all simple or subscripted instance variables u of the same type as t, and all proper states σ the following hold*:

(i) $\sigma(s[u := t]) = \sigma[u := \sigma(t)](s)$,
(ii) $\sigma \models p[u := t]$ *iff* $\sigma[u := \sigma(t)] \models p$.

Proof. By induction on the structure of s and p. □

7. Proof theory for object-oriented programs

We now study (strong) partial correctness of object-oriented programs expressed by *correctness formulas* of the form $\{p\}\ S\ \{q\}$, where S is a program and p and q are assertions. The assertion p is the *precondition* of the correctness formula and q is the *postcondition*. A correctness formula $\{p\}\ S\ \{q\}$ holds in the sense of *partial correctness*, abbreviated $\models \{p\}\ S\ \{q\}$, if every terminating computation of S that starts in a state satisfying p terminates in a state satisfying q. And $\{p\}\ S\ \{q\}$ holds in the sense of *strong partial correctness*, abbreviated $\models_{sp} \{p\}\ S\ \{q\}$, if $\models \{p\}\ S\ \{q\}$ and no computation of S that starts in a state satisfying p ends in a failure.

Using the semantics $\mathcal{M}$ and $\mathcal{M}_{sp}$, we formalize these two interpretations of correctness formulas uniformly as set theoretic inclusions (cf. [3]):

- $\models \{p\}\ S\ \{q\}$ if $\mathcal{M}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$,
- $\models_{sp} \{p\}\ S\ \{q\}$ if $\mathcal{M}_{sp}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$.

Since by definition $\mathbf{fail} \notin [\![q]\!]$ holds, $\mathcal{M}_{sp}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$ implies that S does not fail when started in a proper state σ satisfying p, as required for strong partial correctness.

Example 7.1. Consider again the program $S \equiv \mathbf{this}.find(z)$ of Example 3.1 for finding an object in a linked list. To specify the desired effect of the there declared method *find* we introduce a fresh normal array variable a of type **integer** $\to$ **object** that stores a linked list of objects, as expressed by the assertion

$$p_0 \equiv \forall i \geqslant 0 : a[i].next = a[i+1].$$

We take $p \equiv \mathbf{this} = a[0] \wedge p_0$ as precondition and $q \equiv \exists i \geqslant 0 : z = a[i]$ as postcondition. Then the correctness formula $\{p\}\ S\ \{q\}$ holds in the sense of partial correctness, i.e., upon termination z will store one of the objects in the list. Note that this is a correct specification since the variable a is not used (and hence not changed) in the program S. In general, normal auxiliary array or simple variables have to be used to record the initial values of the program variables.

However, this correctness formula does not hold in the sense of strong partial correctness if the list contains the **null** object before the object stored in the variable z. To avoid this we strengthen the precondition by adding the assertion

$$p_1 \equiv\ \forall i \geqslant 0 : a[i] \neq \mathbf{null}.$$

Then $\{p \wedge p_1\}\ S\ \{q\}$ holds in the sense of strong partial correctness. Finally, if the list is circular and does not contain the **null** object or the object stored in z, the program S diverges.

7.1. Partial correctness

Partial correctness of the programs in the kernel language is proved using the proof system *PK* consisting of the group of axioms and rules 1–8, and 10 shown in Appendix B.1.

We now consider partial correctness of object-oriented programs. First, we introduce the following axiom for assignments to instance variables:

AXIOM 11. ASSIGNMENT TO INSTANCE VARIABLES

$$\big\{p[u := t]\big\}\ u := t\ \{p\}$$

where u is a simple or subscripted instance variable.

So this axiom uses the new substitution operation defined in the previous section. Next, as we shall explain in a moment, we need the following rule for weakening the precondition of a partial correctness formula concerning a method call.

RULE 12. WEAKENING

$$\frac{\{p \wedge s \neq \mathbf{null}\}\ s.m(\bar{t})\ \{q\}}{\{p\}\ s.m(\bar{t})\ \{q\}}.$$

7.1.1. Non-recursive methods

The main issue is how to deal with the parameters of method calls. Therefore, to focus on it we discuss the parameters of non-recursive methods first. The following *copy rule* shows how to prove correctness of non-recursive method calls:

$$\frac{\{p\}\ \textbf{begin local this}, \bar{u} := s, \bar{t};\ S\ \textbf{end}\ \{q\}}{\{p\}\ s.m(\bar{t})\ \{q\}}$$

where $m(\bar{u}) :: S \in D$.

Example 7.2. We prove the partial correctness formula {**true**} **null**.m {**false**}, where $m :: \textbf{skip} \in D$. First, we have

$$\{\textbf{false}\}\ \textbf{begin local this} := \textbf{null}; \textbf{skip end}\ \{\textbf{false}\},$$

so by the above copy rule we get {**false**} **null**.m {**false**}. The desired conclusion now follows by the above weakening rule and the consequence rule.

7.1.2. Recursive methods

When we deal only with one recursive method and use the method call as the considered object-oriented program, the above copy rule needs to be modified to

$$\frac{\{p\}\ s.m(\bar{t})\ \{q\} \vdash_{PO} \{p\}\ \textbf{begin local this}, \bar{u} := s, \bar{t};\ S\ \textbf{end}\ \{q\}}{\{p\}\ s.m(\bar{t})\ \{q\}}$$

where $D = \{m(\bar{u}) :: S\}$.

The provability relation $\vdash_{PO}$ here refers to the proof system *PO*, which is defined as *PK* extended with the axiom 11 for assignments to instance variables, the weakening rule 12, and the auxiliary rules A1–A5 (as introduced in Appendix B.2). Thus the premise of the rule states that in the proof the correctness of the block statement we may *assume* the corresponding correctness formula concerning the method call.

In the case of an arbitrary program and a set of mutually recursive method declarations we have the following generalization of the above rule.

RULE 13. RECURSION I

$$\frac{\begin{array}{l}\{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash_{PO} \{p\}\ S\ \{q\},\\ \{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash_{PO}\\ \quad \{p_i\}\ \textbf{begin local this}, \bar{u}_i := s_i, \bar{t}_i;\ S_i\ \textbf{end}\ \{q_i\},\ i \in \{1, \ldots, n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $m_i(\bar{u}_i) :: S_i \in D$ for $i \in \{1, \ldots, n\}$.

The intuition behind this rule is as follows. Say that a program S is (p, q)-*correct* if $\{p\}\ S\ \{q\}$ holds in the sense of partial correctness. The second premise of the rule states that we can establish from the *assumption* of the (p_i, q_i)-correctness of the method calls $s_i.m_i(\bar{t}_i)$ for $i \in \{1, \ldots, n\}$, the (p_i, q_i)-correctness of the procedure bodies S_i for $i \in \{1, \ldots, n\}$, which are adjusted as in the transition axiom that deals with the method calls. Then we can prove the (p_i, q_i)-correctness of the method calls $s_i.m_i(\bar{t}_i)$ for $i \in \{1, \ldots, n\}$ unconditionally, and thanks to the first premise establish the (p, q)-correctness of the program S.

To prove partial correctness of object-oriented programs we use the following

PROOF SYSTEM PO^+:

This system is obtained by extending *PO* by the recursion I rule 13.

7.2. Strong partial correctness

Strong partial correctness of programs in the kernel language is proved using the proof system *SPK* consisting of the group of axioms and rules 1–7, 9, and 10 shown in Appendix B.1.

To prove strong partial correctness of method calls we modify the above recursion rule I. The provability symbol $\vdash_{SPO}$ refers now to the proof system *SPO*, which is defined as *SPK* augmented with the assignment axiom 11 and the auxiliary rules A1–A5 introduced in Appendix B.2.

RULE 14. RECURSION II

$$\frac{\begin{array}{l}\{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\},\ \ldots,\ \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash_{SPO} \{p\}\ S\ \{q\},\\ \{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\},\ \ldots,\ \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash_{SPO}\\ \quad \{p_i\}\ \textbf{begin local this}, \bar{u}_i := s_i, \bar{t}_i;\ S_i\ \textbf{end}\ \{q_i\},\ i \in \{1,\ldots,n\}\\ (*)\quad p_i \to s_i \neq \textbf{null},\ i \in \{1,\ldots,n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $m_i(\bar{u}_i) :: S_i \in D$, for $i \in \{1,\ldots,n\}$.

Thus compared with the recursion I rule 13, the premises $(*)$ have been added. These premises are indeed needed, as the following incorrect derivation shows.

Example 7.3. Let $m :: \textbf{skip} \in D$. Without the premises $(*)$, we could derive from

$$\{\textbf{true}\}\ \textbf{null}.m\ \{\textbf{true}\} \vdash \{\textbf{true}\}\ \textbf{begin local this} := \textbf{null}; \textbf{skip end}\ \{\textbf{true}\}$$

the correctness formula $\{\textbf{true}\}\ \textbf{null}.m\ \{\textbf{true}\}$. However, this correctness formula does not hold in the sense of strong partial correctness.

To prove strong partial correctness of object-oriented programs we use the following

PROOF SYSTEM SPO^+:

This system is obtained by extending *SPO* by the recursion II rule 14.

8. Formal justification

To prove soundness and completeness of the proof systems *PO* and *SPO* for (strong) partial correctness of object-oriented programs we shall use the transformation given in Section 5, notably the Correctness Theorem 5.4, and reduce the problem to the analysis of the corresponding proof systems for recursive programs.

The *partial correctness semantics* $\mathcal{M}[\![S]\!]$ and the *strong partial correctness semantics* $\mathcal{M}_{sp}[\![S]\!]$ of recursive programs S are defined as for the kernel language. We have the following basic semantic invariance property of recursive programs.

Lemma 8.1 *(Semantic Invariance). Let $\mathcal{N}$ stand for $\mathcal{M}$ or $\mathcal{M}_{sp}$. Further, let $\bar{z}$ be a sequence of fresh variables which do not appear in the main statement S (or the given set of declarations D) and $\bar{d}$ be a corresponding sequence of values. Then*

$$\mathcal{N}[\![S]\!]\big(\sigma[\bar{z} := \bar{d}]\big) = \big\{\tau[\bar{z} := \bar{d}] \bigm| \tau \in \mathcal{N}[\![S]\!](\sigma)\big\}.$$

Proof. The proof proceeds by induction on the length of the computation. □

8.1. Proof theory for recursive programs

Correctness formulas $\{p\}\ S\ \{q\}$ for recursive programs S and their interpretation in terms of partial and strong partial correctness is defined as for object-oriented programs.

In the following rule for recursive programs we use the provability symbol $\vdash$ to refer to either the proof system *PR* which consists of the proof system *PK* augmented with the auxiliary rules A1–A5 defined in Appendix B.2 or the proof system *SPR* which consists of the proof system *SPK* augmented with the these rules.

RULE 15. RECURSION III

$$\frac{\begin{array}{l}\{p_1\}\ P_1(\bar{t}_1)\ \{q_1\},\ \ldots,\ \{p_n\}\ P_n(\bar{t}_n)\ \{q_n\} \vdash \{p\}\ S\ \{q\},\\ \{p_1\}\ P_1(\bar{t}_1)\ \{q_1\},\ \ldots,\ \{p_n\}\ P_n(\bar{t}_n)\ \{q_n\} \vdash\\ \quad \{p_i\}\ \textbf{begin local}\ \bar{u}_i := \bar{t}_i; S_i\ \textbf{end}\ \{q_i\},\ i \in \{1,\ldots,n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $P_i(\bar{u}_i) :: S_i \in D$.

The intuition behind this rule is analogous as in the case of the recursion I rule introduced in Section 7. For recursive programs we use the following proof systems.

PROOF SYSTEM PR^+ for partial correctness of recursive programs:

This system is obtained by extending *PR* by the recursion III rule 15.

PROOF SYSTEM SPR^+ for strong partial correctness of recursive programs:

This system is obtained by extending *SPR* by the recursion III rule 15.

8.2. Translation of assertions and proofs

For the reduction to (correctness proofs of) recursive programs we also have to transform expressions of the assertion language. To this end, we extend the definition of $\Theta(s)$ given in Section 5.2 to global expressions introduced in Section 6.1 by adding the following two cases (where x is an instance variable of basic type and a is an array instance variable):

- $\Theta(s.x) = x[\Theta(s)]$,
- $\Theta(s.a[s_1, \ldots, s_n]) = a[\Theta(s), \Theta(s_1), \ldots, \Theta(s_n)]$.

Then we extend the transformation $\Theta(s)$ to a transformation $\Theta(p)$ of assertions by a straightforward induction on the structure of p. Correctness of this transformation of assertions is stated in the following lemma.

Lemma 8.2 *(Assertion). For all assertions p and all proper states* σ

$$\sigma \models p \quad \textit{iff} \quad \Theta(\sigma) \models \Theta(p).$$

Proof. The straightforward proof proceeds by induction on the structure of p. □

Corollary 8.3 *(Translation I). For all correctness formulas* $\{p\}\ S\ \{q\}$*, where S is an object-oriented program,*

$$\models \{p\}\ S\ \{q\} \quad \textit{iff} \quad \models \big\{\Theta(p)\big\}\ \Theta(S)\ \big\{\Theta(q)\big\},$$

and

$$\models_{sp} \{p\}\ S\ \{q\} \quad \textit{iff} \quad \models_{sp} \big\{\Theta(p)\big\}\ \Theta(S)\ \big\{\Theta(q)\big\}.$$

Proof. It follows directly by the Assertion Lemma 8.2 and the Correctness Theorem 5.4. □

We next show that a correctness proof of an object-oriented program can be translated to a correctness proof of the corresponding recursive program. We first need the following lemma which states equivalence between a correctness proof of a method call from a given set of assumptions and a correctness proof of the corresponding procedure call from the translated set of assumptions. For a given set of assumptions A about method calls, we define the set of assumptions $\Theta(A)$ about the corresponding procedure calls by

$$\Theta(A) = \big\{\{\Theta(p)\}\ m(\Theta(s), \Theta(\bar{t}))\ \{\Theta(q)\} \mid \{p\}\ s.m(\bar{t})\ \{q\} \in A\big\}.$$

Lemma 8.4 *(Translation of Adaptation Correctness Proofs). Let A be a given set of assumptions about method calls. Then*

$$A \vdash_{AR} \{p\}\ s.m(\bar{t})\ \{q\} \quad \textit{iff} \quad \Theta(A) \vdash_{AR} \big\{\Theta(p)\big\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \big\{\Theta(q)\big\},$$

where $\vdash_{AR}$ *denotes provability in the proof system consisting of the so-called* adaptation rules: *the consequence rule* 7 *and the auxiliary proof rules introduced in Appendix* B.2*.*

Proof. The proof proceeds by induction on the length of the derivation. □

In order to prove the equivalence between partial correctness proofs of a method call from a given set of assumptions and correctness proofs of the corresponding procedure call from the translated set of assumptions, we need the following lemma about partial correctness proofs of failure statements.

Lemma 8.5 *(Normal Form Partial Correctness Failure Statements). Let A be a given set of assumptions about procedure calls. If*

$$A \vdash_{PR} \{p\}\ \textbf{if}\ B \to S\ \textbf{fi}\ \{q\}$$

then

$$A \vdash_{PR} \{p \wedge B\}\ S\ \{q\}.$$

Proof. The proof proceeds by induction on the length of the given derivation. By the form of the proof rules we can restrict to the consequence rule 7, the auxiliary proof rules introduced in Appendix B.2, and the failure rule 8. We consider the case of an application of the auxiliary rule A3. Let

$$A \vdash_{PR} \{p'\}\ \textbf{if}\ B \to S\ \textbf{fi}\ \{q\}$$

and p denote $\exists x : p'$, where $x \notin Var(D) \cup Var(S) \cup free(q)$. By the induction hypothesis and an application of the auxiliary rule 8, we have

$$A \vdash_{PR} \{\exists x : (p' \wedge B)\}\ S\ \{q\}.$$

Since x does not occur in B, the precondition is logically equivalent to $(\exists x : p') \wedge B$, so the desired result follows by an application of the consequence rule. □

Next, we introduce the following lemmas stating the equivalence between (strong) partial correctness proofs of a method call from a given set of assumptions and correctness proofs of the corresponding procedure call from the translated set of assumptions.

Lemma 8.6 *(Translation of Partial Correctness Proofs). Let A be a given set of assumptions about method calls. Then*

$$A \vdash_{PO} \{p\}\ s.m(\bar{t})\ \{q\}$$

iff

$$\Theta(A) \vdash_{PR} \{\Theta(p)\}\ \textbf{if}\ \Theta(s) \neq \textbf{null} \to m\big(\Theta(s), \Theta(\bar{t})\big)\ \textbf{fi}\ \{\Theta(q)\}.$$

Proof. Note that by the form of the proof rules we can restrict the rules of *PO* to the proof system *AR* extended with the weakening rule 12 and restrict the rules of *PR* to the proof system *AR* extended with the failure rule 8.

($\Rightarrow$) We prove the claim by induction on the length of the derivation. For the base case assume that $\{p\}\ s.m(\bar{t})\ \{q\} \in A$. By definition of $\Theta(A)$,

$$\{\Theta(p)\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \{\Theta(q)\} \in \Theta(A),$$

so

$$\Theta(A) \vdash_{PR} \{\Theta(p)\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \{\Theta(q)\}.$$

By a trivial application of the consequence rule, we get

$$\Theta(A) \vdash_{PR} \{\Theta(p) \wedge \Theta(s) \neq \textbf{null}\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \{\Theta(q)\}.$$

Now by the failure rule, we get the desired result.

For the induction step we treat the case when the last rule applied is the weakening rule. Then it is applied to

$$A \vdash_{PO} \{p \wedge s \neq \textbf{null}\}\ s.m(\bar{t})\ \{q\}.$$

By the induction hypothesis,

$$\Theta(A) \vdash_{PR} \{\Theta(p) \wedge \Theta(s) \neq \textbf{null}\}\ \textbf{if}\ \Theta(s) \neq \textbf{null} \to m\big(\Theta(s), \Theta(\bar{t})\big)\ \textbf{fi}\ \{\Theta(q)\}.$$

By Lemma 8.5, it follows that

$$\Theta(A) \vdash_{PR} \{\Theta(p) \wedge \Theta(s) \neq \textbf{null} \wedge \Theta(s) \neq \textbf{null}\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \{\Theta(q)\},$$

so by the consequence and failure rules we get the desired result, the right-hand side of the statement of the lemma.

($\Leftarrow$) We prove the claim by induction on the length of the derivation. We only treat the main case of the induction step when the last rule applied is the failure rule. Then it is applied to

$$\Theta(A) \vdash_{PR} \{\Theta(p) \wedge \Theta(s) \neq \textbf{null}\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \{\Theta(q)\}.$$

In this derivation in $\vdash_{PR}$ the failure rule has not been applied. Thus we can replace $\vdash_{PR}$ by $\vdash_{AR}$. By the Translation Lemma 8.4, we get

$$A \vdash_{AR} \{p \wedge s \neq \textbf{null}\}\ s.m(\bar{t})\ \{q\}.$$

Applying the weakening rule we get the desired result, the left-hand side of the statement of the lemma. □

Lemma 8.7 *(Translation of Strong Partial Correctness Proofs). Let A be a given set of assumptions about method calls such that* $p' \to s' \neq \textbf{null}$ *holds for all* $\{p'\}\ s'.m'(\bar{t}')\ \{q'\} \in A$. *Then*

$$A \vdash_{SPO} \{p\}\ s.m(\bar{t})\ \{q\}$$

iff

$$\Theta(A) \vdash_{SPR} \{\Theta(p)\}\ \textbf{if}\ \Theta(s) \neq \textbf{null} \to m\big(\Theta(s), \Theta(\bar{t})\big)\ \textbf{fi}\ \{\Theta(q)\}.$$

Proof. ($\Rightarrow$) We prove the claim by induction on the length of the derivation. We only treat the base case, that is when $\{p\}\ s.m(\bar{t})\ \{q\} \in A$. By definition of A, the implication $p \to s \neq \textbf{null}$ holds. By definition of $\Theta(A)$,

$$\big\{\Theta(p)\big\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \big\{\Theta(q)\big\} \in \Theta(A).$$

Furthermore, by the Assertion Lemma 8.2, we have $\Theta(p) \to \Theta(s) \neq \textbf{null}$. So we conclude the desired result by an application of the failure II rule.

($\Leftarrow$) We prove the claim by induction on the length of the derivation. We only treat the main case of the inductive step, an application of the failure II rule. So $\Theta(p) \to \Theta(s) \neq \textbf{null}$ holds and the rule is applied to

$$\Theta(A) \vdash_{SPR} \big\{\Theta(p)\big\}\ m\big(\Theta(s), \Theta(\bar{t})\big)\ \big\{\Theta(q)\big\}.$$

In this derivation in $\vdash_{SPR}$ the failure II rule has not been applied. So we can replace $\vdash_{SPR}$ by $\vdash_{AR}$. Thus by the Translation Lemma 8.4,

$$A \vdash_{AR} \{p\}\ s.m(\bar{t})\ \{q\}$$

from which the desired result follows. □

In order to extend the above lemmas from method calls to arbitrary statements we need the following lemma which states that the transformation on assertions is a homomorphism with respect to the substitution operation.

Lemma 8.8 *(Homomorphism). For all global expressions or assertions p, all expressions t of the programming language, and all simple or subscripted variables u,*

$$\Theta\big(p[u := t]\big) \equiv \Theta(p)\big[\Theta(u) := \Theta(t)\big].$$

Proof. We treat the case of a global expression s and a simple instance variable u. By definition, $\Theta(u) \equiv u[\textbf{this}]$. It suffices to prove

$$\Theta\big(s[u := t]\big) \equiv \Theta(s)\big[u[\textbf{this}] := \Theta(t)\big]$$

by induction on the structure of the global expression s. We treat the case of $s \equiv e.u$.

$$\begin{aligned}
&\Theta\big(e.u[u := t]\big)\\
\equiv\ &\big\{\text{by definition of the substitution } [u := t]\big\}\\
&\Theta\big(\textbf{if } e[u := t] = \textbf{this then } t \textbf{ else } e[u := t].u \textbf{ fi}\big)\\
\equiv\ &\{\text{by definition of } \Theta\}\\
&\textbf{if } \Theta\big(e[u := t] = \textbf{this}\big) \textbf{ then } \Theta(t) \textbf{ else } \Theta\big(e[u := t].u\big) \textbf{ fi}\\
\equiv\ &\{\text{by definition of } \Theta\}\\
&\textbf{if } \Theta\big(e[u := t]\big) = \textbf{this then } \Theta(t) \textbf{ else } u\big[\Theta\big(e[u := t]\big)\big] \textbf{ fi}\\
\equiv\ &\{\text{by induction hypothesis about } e\}\\
&\textbf{if } \Theta(e)\big[u[\textbf{this}] := \Theta(t)\big] = \textbf{this then } \Theta(t) \textbf{ else } u\big[\Theta(e)\big[u[\textbf{this}] := \Theta(t)\big]\big] \textbf{ fi}\\
\equiv\ &\big\{\text{by definition of the substitution } \big[u[\textbf{this}] := \Theta(t)\big]\big\}\\
&u\big[\Theta(e)\big]\big[u[\textbf{this}] := \Theta(t)\big]\\
\equiv\ &\{\text{by definition of } \Theta\}\\
&\Theta(e.u)\big[u[\textbf{this}] := \Theta(t)\big]. \qquad \square
\end{aligned}$$

Lemma 8.9 *(Translation of Correctness Proofs Statements). Let A be a set of assumptions about method calls and $\{p\}\ S\ \{q\}$ be a correctness formula of an object-oriented statement S. Then*

$$A \vdash \{p\}\ S\ \{q\} \quad \textit{iff} \quad \Theta(A) \vdash \big\{\Theta(p)\big\}\ \Theta(S)\ \big\{\Theta(q)\big\},$$

where

- *in case of partial correctness $\vdash$ on the left-hand side denotes provability in the proof system PO, and $\vdash$ on the right-hand side denotes provability in the proof system PR, and*
- *in case of strong partial correctness $\vdash$ on the left-hand side denotes provability in the proof system SPO, and $\vdash$ on the right-hand side denotes provability in the proof system SPR. Additionally, we assume that $p' \to s \neq \textbf{null}$ holds for all $\{p'\}\ s.m(\bar{t})\ \{q'\} \in A$.*

Proof. The proof proceeds by induction on the length of the derivation. The case of an assignment statement follows by the Homomorphism Lemma 8.8. The case of a method call follows by the Translation Lemma 8.7. The cases of other program statements follow directly by the induction hypothesis. In particular, in the cases of the consequence rule and the rules for conditionals and loops, the Assertion Lemma 8.2 is used. □

Finally, we arrive at the main result of this section.

Theorem 8.10 *(Translation II). For all correctness formulas $\{p\}\ S\ \{q\}$, where S is an object-oriented program,*

(i) *$\{p\}\ S\ \{q\}$ is derivable in the proof system PO^+ iff $\{\Theta(p)\}\ \Theta(S)\ \{\Theta(q)\}$ is derivable in PR^+,*
(ii) *$\{p\}\ S\ \{q\}$ is derivable in the proof system SPO^+ iff $\{\Theta(p)\}\ \Theta(S)\ \{\Theta(q)\}$ is derivable in SPR^+.*

Proof. The proof proceeds by an induction on the length of the derivation. The case of the assignment axioms is taken care of by the above Lemma 8.8. The case of the recursion rules is taken care of by the Translation Lemma 8.9. The case of the other axioms and rules follows immediately from the induction hypothesis (using the Assertion Lemma 8.2 in case of the rules for the conditional and while statements). Note that in the premises of the recursion rules we cannot apply the recursion rule again. □

From the above theorem it immediately follows that the proof systems PO^+ and SPO^+ are *sound* and (relative) *complete* if and only if the corresponding proof systems PR^+ and SPR^+ are sound and (relative) complete. For proofs of soundness of the systems PR^+ and SPR^+, that is, for every correctness formula $\{p\}\ S\ \{q\}$ about a recursive program S, derivability of $\{p\}\ S\ \{q\}$ in PR^+ and SPR^+ implies $\models \{p\}\ S\ \{q\}$ and $\models_{sp} \{p\}\ S\ \{q\}$, respectively, we refer to our book [3]. In the next section we discuss (relative) completeness of the proof systems PR^+ and SPR^+.

9. Completeness

We prove here relative completeness of the proof systems PR^+ and SPR^+ for partial and strong partial correctness of the class of recursive programs considered in this paper. The proof is based on the use of *weakest preconditions*. As explained in Section 10, this approach also applies to total correctness. We first discuss the expressibility of weakest preconditions for recursive programs that use variables whose type may involve *abstract data types* (like the basic type **object**).

9.1. Expressibility

We introduce the following definitions and conventions. By $\sigma =_V \sigma'$, for $V \subseteq Var$, we denote the fact that $\sigma(v) = \sigma'(v)$, for $v \in V$. We fix throughout this section a sequence $\bar{x} = x_1, \ldots, x_k$ of (simple and array) variables and a main statement S such that its variables and those of the given set of declarations D are contained in $\bar{x}$. Further, we fix a corresponding sequence $\bar{y}$ of fresh variables used to refer to the *final* values of $\bar{x}$ in the definition of the weakest preconditions below. By $\bar{x} = \bar{y}$ we denote the conjunction of the formulas $x_i = y_i$, for a simple variable x_i, and $\forall \bar{u}_i : x_i[\bar{u}_i] = y_i[\bar{u}_i]$, for an array variable x_i ($\bar{u}_i$ denotes a sequence of simple variables corresponding to the argument types of x_i). The update $\sigma[\bar{x} := \sigma'(\bar{y})]$ assigns to each variable x_i the value/function $\sigma'(y_i)$, for $i \in \{1, \ldots, k\}$. We denote by the substitution $p[\bar{x} := \bar{y}]$ the result of *renaming* every variable x_i by y_i, for $i \in \{1, \ldots, k\}$. We have the following substitution lemma corresponding to the Substitution Lemma 6.4.

Lemma 9.1 *(Substitution). We have*

$$\sigma \models p[\bar{x} := \bar{y}] \quad \textit{iff} \quad \sigma\big[\bar{x} := \sigma(\bar{y})\big] \models p.$$

Proof. The proof proceeds by induction on the structure of p. □

The weakest precondition $WP(S, p)$ for *partial* correctness denotes the set

$$\big\{\sigma \;\big|\; \mathcal{M}[\![S]\!](\sigma) \subseteq [\![p]\!]\big\}.$$

Similarly, the weakest precondition $WP_{sp}(S, p)$ for *strong* partial correctness denotes the set

$$\big\{\sigma \;\big|\; \mathcal{M}_{sp}[\![S]\!](\sigma) \subseteq [\![p]\!]\big\}.$$

The above predicates satisfy the following equations.

Lemma 9.2 *(Weakest Precondition Calculus). Let W stand for WP or WP_{sp}. The weakest preconditions satisfy the following (standard) equations.*

- $W(\mathit{skip}, p) = [\![p]\!]$,
- $W(u := t, p) = [\![p[u := t]]\!]$,
- $W(\bar{x} := \bar{t}, p) = [\![p[\bar{x} := \bar{t}]]\!]$,
- $W(S_1; S_2, p) = W(S_1, W(S_2, p))$,
- $W(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, p) = ([\![B]\!] \cap W(S_1, p)) \cup ([\![\neg B]\!] \cap W(S_2, p))$,
- $W(\textbf{while } B \textbf{ do } S \textbf{ od}, p) = ([\![\neg B]\!] \cap [\![p]\!]) \cup ([\![B]\!] \cap W(S, W(\textbf{while } B \textbf{ do } S \textbf{ od}, p)))$.

Failure statements satisfy

$$WP(\textbf{if } B \to S \textbf{ fi}, p) = \big([\![B]\!] \cap WP(S, p)\big) \cup [\![\neg B]\!]$$

and

$$WP_{sp}(\textbf{if } B \to S \textbf{ fi}, p) = [\![B]\!] \cap WP_{sp}(S, p).$$

Finally, block statements satisfy

$$W(\textbf{begin local } \bar{u} := \bar{t}; S \textbf{ end}, p) = W(\bar{u} := \bar{t}; S, p),$$

where the local variables $\bar{u}$ do not appear in p.

Proof. We prove the equation for block statements (the equations for the other statements are standard). By definition of the semantics of block statements and the above equations for (parallel) assignments and sequential composition of statements, we have

$$\begin{aligned} &W(\textbf{begin local } \bar{u} := \bar{t}; S \textbf{ end}, p) \\ &\quad = W\big(\bar{u} := \bar{t}; S; \bar{u} := \sigma(\bar{u}), p\big) \\ &\quad = W\big(\bar{u} := \bar{t}; S; W\big(\bar{u} := \sigma(\bar{u}), p\big)\big) \\ &\quad = W(\bar{u} := \bar{t}; S, p). \end{aligned}$$

Note that $p[\bar{u} := \sigma(\bar{u})]$ equals p because the variables $\bar{u}$ do not appear in p. □

As a special case, we introduce the following *most general weakest preconditions*

$$WP(S, \bar{x} = \bar{y}) \quad \text{and} \quad WP_{sp}(S, \bar{x} = \bar{y}).$$

Note that by definition,

$$WP(S, \bar{x} = \bar{y}) = \big\{\sigma \;\big|\; \mathcal{M}[\![S]\!](\sigma) \subseteq \big\{\sigma\big[\bar{x} := \sigma(\bar{y})\big]\big\}\big\}$$

and

$$WP_{sp}(S, \bar{x} = \bar{y}) = \big\{\sigma \;\big|\; \mathcal{M}_{sp}[\![S]\!](\sigma) \subseteq \big\{\sigma\big[\bar{x} := \sigma(\bar{y})\big]\big\}\big\}.$$

These predicates describe the *graphs* of the *deterministic* functions $\mathcal{M}[\![S]\!]$ and $\mathcal{M}_{sp}[\![S]\!]$ in terms of a relation between the input variables $\bar{x}$ and the output variables $\bar{y}$.

In order to express these most general weakest preconditions in the *first-order* assertion language we introduce a *state-based* encoding of the basic types which allows for a standard arithmetic encoding of the programming semantics.

Let **nat** denote the basic type of the set $\mathbb{N}$ of natural numbers. For each basic type T we fix a fresh array variable h_T of type $\textbf{nat} \to T$ for a state-based encoding of the values of basic type T. We will use h to range over the variables h_T. Without loss of generality we restrict our attention to states for which the interpretation of each array variable h_T specifies an *enumeration* of the values of the basic type T, that is, h_T is *surjective*, as expressed by

$$\forall x : \exists n : x = h_T[n],$$

where x is of type T and n of type **nat**.

Given this encoding of the basic types, we next show how to express in the assertion language the encoding of the interpretation of the variables. The assertion $\mathit{code}(n, z)$ defined by $h[n] = z$, where z is a (simple) variable, directly expresses that the variable n (of type **nat**) stores an integer representation of the value of z. In order to express a similar assertion $\mathit{code}(n, a)$, where a is an array variable, we assume in the assertion language the following arithmetic operations:

- $\langle \bar{n} \rangle$ denotes the natural number encoding the sequence of natural numbers $\bar{n}$,
- $n(i)$ denotes the ith element of the sequence encoded by n,
- $|n|$ denotes the length of the sequence of numbers encoded by n.

We note that the above operations can be formally defined in the assertion language by some *computable* enumeration of all finite sequences of natural numbers (details are standard and therefore omitted). Let l denote the number of argument types of the array variable a. The assertion $app(n, a)$ defined by

$$|n| = l + 1 \wedge a\big[h\big[n(1)\big], \ldots, h\big[n(l)\big]\big] = h\big[n(l+1)\big]$$

expresses that n encodes an application of the interpretation of the array a. The assertion $code(n, a)$ is then defined by

$$\bigwedge_{k=1}^{|n|} app\big(n(k), a\big).$$

This assertion expresses that n encodes a finite sequence of numbers $n(k)$ each of which in turn encodes an application of the interpretation of the array a. For every sequence $\bar{z} = z_1, \ldots, z_k$ of variables we denote by $code(n, \bar{z})$ the conjunction

$$|n| = k \wedge \bigwedge_{i=1}^{k} code\big(n(i), z_i\big).$$

Without loss of generality we assume that the encoding of any sequence of variables $\bar{z}$ is *surjective*: for every $n \in \mathbb{N}$ there exists a state σ such $\sigma \models code(n, \bar{z})$.

Given this encoding of the interpretation of simple and array variables, we next introduce the following binary *arithmetic* relation $comp_S$, where S is a recursive program, which denotes the set

$$\big\{(n, m) \;\big|\; \forall \sigma : \sigma \models code(n, \bar{x}) \text{ and } \sigma \models code^+(m, \bar{x}, \bar{y}) \text{ implies } \sigma\big[\bar{x} := \sigma(\bar{y})\big] \in \mathcal{M}[\![S]\!](\sigma)\big\}.$$

Here we implicitly assume that $code^+(m, \bar{x}, \bar{y})$ asserts $code(m, \bar{y})$ and *additionally* enforces that each array variable x_i agrees with the corresponding array variable y_i on the *complement* of the domain specified by $m(i)$ (note that m codes only a *finite* part of each array variable of $\bar{y}$). (The details of this extension of the assertion $code(m, \bar{y})$ are straightforward though somewhat tedious and therefore omitted.) In the sequel we write $comp_S(n, m)$ to denote that n and m belong to the binary relation $comp_S$ (we will use n and m both to denote natural numbers and variables of type **nat**).

Since every finite computation of S accesses each array variable of $\bar{x}$ only on a finite subset of the domain of its interpretation, we have the following closure property of $comp_S$.

Lemma 9.3 *(Closure of comp). For all states σ and σ' we have that $\sigma' \in \mathcal{M}[\![S]\!](\sigma)$ implies $comp_S(n, m)$, for some pair of numbers n and m such that $\sigma \models code(n, \bar{x})$ and $\sigma[\bar{y} := \sigma'(\bar{x})] \models code^+(m, \bar{x}, \bar{y})$.*

We proceed with the introduction of the (unary) arithmetic predicate $fail_S$ which denotes the set

$$\big\{n \;\big|\; \forall \sigma : \sigma \models code(n, \bar{x}) \text{ implies } \mathbf{fail} \in \mathcal{M}_{sp}[\![S]\!](\sigma)\big\}.$$

In the sequel we also use $fail_S(n)$ to denote that n is an element of the set $fail_S$.

Again, since every finite sequence of computation steps of S accesses each array variable of $\bar{x}$ only on a finite subset of the domain of its interpretation, we can assume the following closure property of the $fail_S$.

Lemma 9.4 *(Closure of fail). For every state σ such that $\mathbf{fail} \in \mathcal{M}_{sp}[\![S]\!](\sigma)$ we have $\sigma \models code(n, \bar{x})$ and $fail_S(n)$, for some natural number n.*

By means of standard techniques for encoding finite sequences of computation steps (see for example [11]) we can express the predicates $comp_S$ and $fail_S$ arithmetically in the (first-order) assertion language in terms of the above encoding of the interpretation of the variables $\bar{x}$. Therefore, we may assume without loss of generality that these predicates are present in the assertion language.

Lemma 9.5 *(Expressibility). For the assertion*

$$p \;\equiv\; \forall n, m : \big(code(n, \bar{x}) \wedge comp_S(n, m)\big) \to code(m, \bar{y})$$

we have

$$WP(S, \bar{x} = \bar{y}) = [\![p]\!]$$

and

$$WP_{sp}(S, \bar{x} = \bar{y}) = \big[\!\big[p \wedge \forall n : code(n, \bar{x}) \to \neg fail_S(n)\big]\!\big].$$

Proof. We prove first the first equation. Let $\sigma \in WP(S, \bar{x} = \bar{y})$, i.e., $\mathcal{M}[\![S]\!](\sigma) \subseteq \{\sigma[\bar{x} := \sigma(\bar{y})]\}$. In order to prove $\sigma \models p$, let $\sigma \models code(n, \bar{x})$ and $comp_S(n, m)$, for some arbitrary (constants) n and m. Further, let $\sigma' \models code(m, \bar{y})$, for some σ' (note that the encoding of the variables $\bar{y}$ is assumed to be surjective). Without loss of generality we may assume that $\sigma[\bar{y} := \sigma'(\bar{y})] \models code^+(m, \bar{x}, \bar{y})$ (note that m only codes a finite part of σ'). Since the evaluation of $code(n, \bar{x})$ only depends on the interpretation of the variables $\bar{x}$, $\sigma \models code(n, \bar{x})$ implies $\sigma[\bar{y} := \sigma'(\bar{y})] \models code(n, \bar{x})$. By definition of $comp_S$ it follows that $\sigma[\bar{x} := \sigma'(\bar{y})] \in \mathcal{M}[\![S]\!](\sigma)$ (note that $\bar{y}$ are assumed not to occur in S). Because S is deterministic it follows that $\sigma[\bar{x} := \sigma'(\bar{y})] = \sigma[\bar{x} := \sigma(\bar{y})]$, i.e., $\sigma'(\bar{y}) = \sigma(\bar{y})$. So we conclude that $\sigma \models code(m, \bar{y})$ (note that $code^+(m, \bar{x}, \bar{y})$ trivially implies $code(m, \bar{y})$).

Next let $\sigma \models p$. In order to prove $\sigma \in WP(S, \bar{x} = \bar{y})$, i.e., $\mathcal{M}[\![S]\!](\sigma) \subseteq \{\sigma[\bar{x} := \sigma(\bar{y})]\}$, let $\sigma' \in \mathcal{M}[\![S]\!](\sigma)$. By Lemma 9.3, it follows that $comp_S(n, m)$, for some n and m such that $\sigma \models code(n, \bar{x})$ and $\sigma[\bar{y} := \sigma'(\bar{x})] \models code^+(m, \bar{x}, \bar{y})$. So by the definition of $comp_S$ it follows that $\sigma[\bar{x} := \sigma'(\bar{x})] \in \mathcal{M}[\![S]\!](\sigma)$ (as above, note that $\sigma \models code(n, \bar{x})$ implies $\sigma[\bar{y} := \sigma'(\bar{x})] \models code(n, \bar{x})$ and the variables $\bar{y}$ do not appear in S). Since S is deterministic we conclude that $\sigma'(\bar{x}) = \sigma(\bar{y}) = \sigma'(\bar{y})$.

For the second equation, it suffices to observe that by Lemma 9.4, $\mathbf{fail} \in \mathcal{M}_{sp}[\![S]\!](\sigma)$ implies $\sigma \models \exists n : code(n, \bar{x}) \wedge fail_S(n)$. On the other hand, by definition of $fail_S$ it immediately follows that $\sigma \models \exists n : code(n, \bar{x}) \wedge fail_S(n)$ implies $\mathbf{fail} \in \mathcal{M}_{sp}[\![S]\!](\sigma)$. We conclude that $\mathbf{fail} \notin \mathcal{M}_{sp}[\![S]\!](\sigma)$ iff $\sigma \models \forall n : code(n, \bar{x}) \to \neg fail_S(n)$. □

We conclude this discussion of the encoding of the most general weakest preconditions with the following characterization of divergence or failure, in case of partial correctness, and divergence, in case of strong partial correctness.

Lemma 9.6 *(Expressibility of Divergence/Failure). For the assertion*

$$p \equiv \forall n, m : code(n, \bar{x}) \to \neg comp_S(n, m)$$

we have

$$WP(S, \mathbf{false}) = [\![p]\!],$$

and

$$WP_{sp}(S, \mathbf{false}) = \big[\![p \wedge \forall n : code(n, \bar{x}) \to \neg fail_S(n)\big]\!].$$

The formula p in the first equality expresses all states from which S can diverge or fail, while the formula on the right-hand side of the second equality expresses all states from which S can diverge.

Proof. We prove the first equation (the second is dealt with as above). First let $\sigma \in WP(S, \mathbf{false})$, i.e., $\mathcal{M}[\![S]\!](\sigma) = \emptyset$. In order to prove $\sigma \models p$, let $\sigma \models code(n, \bar{x})$ and $comp_S(n, m)$, for some arbitrary (constants) n and m. As above, we may assume without loss of generality that $\sigma[\bar{y} := \sigma'(\bar{x})] \models code^+(m, \bar{x}, \bar{y})$, for some σ'. By definition of $comp_S$, it follows that $\sigma[\bar{x} := \sigma'(\bar{x})] \in \mathcal{M}[\![S]\!](\sigma)$ which contradicts $\mathcal{M}[\![S]\!](\sigma) = \emptyset$.

Next let $\sigma \models p$. In order to prove $\sigma \in WP(S, \mathbf{false})$, i.e., $\mathcal{M}[\![S]\!](\sigma) = \emptyset$, let $\sigma' \in \mathcal{M}[\![S]\!](\sigma)$. By Lemma 9.3, it follows that $comp_S(n, m)$, for some n and m such that $\sigma \models code(n, \bar{x})$ (and $\sigma[\bar{y} := \sigma'(\bar{x})] \models code^+(m, \bar{x}, \bar{y})$). But this contradicts $\sigma \models p$. So we conclude that $\mathcal{M}[\![S]\!](\sigma) = \emptyset$. □

9.2. Completeness proof using most general correctness formulas

We prove (relative) completeness of the proof system SPR^+, i.e., every strong partially correct specification $\{p\}\ S\ \{q\}$ of a recursive program S is derivable in SPR^+. Formally, $\models_{sp} \{p\}\ S\ \{q\}$ implies $\vdash_{SPR^+} \{p\}\ S\ \{q\}$. The proof of (relative) completeness of the proof system PR^+ for partial correctness of recursive programs is similar.

First we state and prove the following completeness result for the most general correctness formulas. Its formulation refers to the expressibility of the most general weakest preconditions justified by the Expressibility Lemma 9.5.

Lemma 9.7 *(Completeness: Most General Correctness Formulas). Let $\bar{x} = x_1, \ldots, x_k$ be all the variables (global and local) appearing in D, S, p or q, and $\bar{y}$ be a corresponding sequence of fresh variables. Further, let q be a consistent assertion, i.e., $[\![q]\!] \neq [\![\mathbf{false}]\!]$. We have*

$$\models_{sp} \{p\}\ S\ \{q\} \quad \textit{implies} \quad \big\{WP_{sp}(S, \bar{x} = \bar{y})\big\}\ S\ \{\bar{x} = \bar{y}\} \vdash_{SPR}\ \{p\}\ S\ \{q\}.$$

Proof. Let $\models_{sp} \{p\}\ S\ \{q\}$, with q a consistent assertion. Without loss of generality we may assume that p and q do not refer to the variables $\bar{y}$ (otherwise, we rename them and apply the substitution rule). Applying the invariance rule we obtain

$$\big\{q[\bar{x} := \bar{y}] \wedge WP_{sp}(S, \bar{x} = \bar{y})\big\}\ S\ \big\{q[\bar{x} := \bar{y}] \wedge \bar{x} = \bar{y}\big\}.$$

Clearly the postcondition implies q. By the consequence rule, we then obtain

$$\big\{q[\bar{x} := \bar{y}] \wedge WP_{sp}(S, \bar{x} = \bar{y})\big\}\ S\ \{q\}.$$

Next we apply the auxiliary rule A3:

$$\{\exists \bar{y} : q[\bar{x} := \bar{y}] \wedge WP_{sp}(S, \bar{x} = \bar{y})\}\ S\ \{q\}.$$

By definition of the weakest precondition and $\models_{sp} \{p\}\ S\ \{q\}$, it follows that p implies the above precondition (an application of the consequence rule thus gives us the desired correctness formula): Let $\sigma \models p$. It follows from $\models_{sp} \{p\}\ S\ \{q\}$ that $\mathcal{M}_{sp}[\![S]\!](\sigma) \subseteq [\![q]\!]$, i.e., $\mathcal{M}_{sp}[\![S]\!](\sigma) = \emptyset$ or $\sigma' \in \mathcal{M}_{sp}[\![S]\!](\sigma)$, for some proper state σ'. First we consider the case that $\mathcal{M}_{sp}[\![S]\!](\sigma) = \emptyset$. From Lemma 8.1 it follows that $\mathcal{M}_{sp}[\![S]\!](\sigma') = \emptyset$, for *every* σ' such $\sigma' =_{Var\setminus \bar{y}} \sigma$. Further, since q is consistent, we have $\sigma' \models q[\bar{x} := \bar{y}]$, for *some* σ' such $\sigma' =_{Var\setminus \bar{y}} \sigma$. Summarizing, we have

$$\sigma' \models q[\bar{x} := \bar{y}] \wedge WP_{sp}(S, \bar{x} = \bar{y}),$$

for some σ' such that $\sigma' =_{Var\setminus \bar{y}} \sigma$. From which we conclude that

$$\sigma \models \exists \bar{y} : q[\bar{x} := \bar{y}] \wedge WP_{sp}(S, \bar{x} = \bar{y}).$$

Next we consider the case that $\sigma' \in \mathcal{M}_{sp}[\![S]\!](\sigma)$, for some proper state σ'. From Lemma 8.1 it follows that $\sigma' \in \mathcal{M}_{sp}[\![S]\!](\sigma)$ implies $\sigma'[\bar{y} := \sigma'(\bar{x})] \in \mathcal{M}_{sp}[\![S]\!](\sigma[\bar{y} := \sigma'(\bar{x})])$. Clearly, $\sigma'[\bar{y} := \sigma'(\bar{x})] \models \bar{x} = \bar{y}$, and therefore $\sigma[\bar{y} := \sigma'(\bar{x})] \models WP_{sp}(S, \bar{x} = \bar{y})$. Further, since $\bar{y}$ are assumed not to appear in p, $\sigma \models p$ implies $\sigma[\bar{y} := \sigma'(\bar{x})] \models p$. So we derive from the assumption $\models_{sp} \{p\}\ S\ \{q\}$ that $\sigma'[\bar{y} := \sigma'(\bar{x})] \models q$. By Lemma 9.1 it then follows that $\sigma'[\bar{y} := \sigma'(\bar{x})] \models q[\bar{x} := \bar{y}]$. Since $\sigma =_{Var\setminus \bar{x}} \sigma'$ and the evaluation of $q[\bar{x} := \bar{y}]$ does not depend on the interpretation of the variables $\bar{x}$, we have also $\sigma[\bar{y} := \sigma'(\bar{x})] \models q[\bar{x} := \bar{y}]$. Summarizing, we obtain that

$$\sigma\big[\bar{y} := \sigma'(\bar{x})\big] \models q[\bar{x} := \bar{y}] \wedge WP_{sp}(S, \bar{x} = \bar{y}),$$

that is,

$$\sigma \models \exists \bar{y} : q[\bar{x} := \bar{y}] \wedge WP_{sp}(S, \bar{x} = \bar{y}). \qquad \square$$

We next introduce for each procedure call $P_i(\bar{t}_i)$, $i \in \{1, \ldots, n\}$, appearing in the given set of declarations D or the main statement S, the correctness formulas

$$\{WP_{sp}\big(P_i(\bar{t}_i), \mathbf{false}\big)\}\ P_i(\bar{t}_i)\ \{\mathbf{false}\} \quad \text{and} \quad \{WP_{sp}\big(P_i(\bar{t}_i), \bar{x} = \bar{y}\big)\}\ P_i(\bar{t}_i)\ \{\bar{x} = \bar{y}\},$$

where $\bar{x} = x_1, \ldots, x_k$ are all variables (global and local) appearing in D or S, and $\bar{y}$ is a corresponding sequence of fresh variables. We rely here on the expressibility of divergence and failure, as justified by the Expressibility Lemma 9.6. Let A denote the set of these correctness formulas.

Lemma 9.8 *(Completeness Assumptions A). We have*

$$\models_{sp} \{p\}\ S\ \{q\} \quad \textit{implies} \quad A \vdash_{SPR} \{p\}\ S\ \{q\}.$$

Proof. The proof proceeds by induction on the structure of the statement S. Distinguishing between $\models_{sp} \{p\}\ S\ \{\mathbf{false}\}$ and $\models_{sp} \{p\}\ S\ \{q\}$, where q is consistent, by definition of $WP_{sp}(S, \mathbf{false})$ and the above Lemma 9.7, it suffices to prove

$$A \vdash_{SPR} \{WP_{sp}(S, r)\}\ S\ \{r\},$$

where r denotes the assertion **false** or $\bar{x} = \bar{y}$. For assignments, sequential composition of statements, conditionals, failure statements and while statements the derivability of these correctness formulas follow from the standard properties of weakest preconditions as described in Lemma 9.2. We consider therefore the non-standard case that S denotes a block statement **begin local** $\bar{u} := \bar{t}; S_1$ **end**. We introduce a sequence $\bar{z}$ of fresh variables corresponding to the local variables $\bar{u}$. By the semantics of block statements, it follows that

$$\models_{sp} \{\bar{z} = \bar{u} \wedge WP_{sp}(S, r)\}\ \bar{u} := \bar{t}; S_1\ \{r[\bar{u} := \bar{z}]\}.$$

By the (general) induction hypothesis, we can derive this correctness formula from the given set of assumptions A. Next we apply the block rule which gives

$$\{\bar{z} = \bar{u} \wedge WP_{sp}(S, r)\}\ S\ \{r[\bar{u} := \bar{z}]\}.$$

We proceed by an application of the invariance rule, which gives us

$$\{\bar{z} = \bar{u} \wedge WP_{sp}(S, r)\}\ S\ \{\bar{z} = \bar{u} \wedge r[\bar{u} := \bar{z}]\}.$$

The postcondition clearly implies r, so by the consequence rule, we obtain

$$\{\bar{z} = \bar{u} \wedge WP_{sp}(S, r)\}\ S\ \{r\}.$$

Finally, applying the substitution rule (replacing in the precondition $\bar{z}$ by $\bar{u}$) followed a trivial application of the consequence rule gives us the desired result. $\square$

We conclude with the following main completeness theorem.

Theorem 9.9 *(Completeness: Strong Partial Correctness). Every strong partially correct specification* $\{p\}\ S\ \{q\}$ *of a recursive program* S *is derivable in* SPR^+. *Formally,* $\models_{sp} \{p\}\ S\ \{q\}$ *implies* $\vdash_{SPR^+} \{p\}\ S\ \{q\}$.

Proof. Let $\models_{sp} \{p\}\ S\ \{q\}$ and A be the set of assumptions as defined above. By Lemma 9.8, we have

$$A \vdash_{SPR} \{p\}\ S\ \{q\}.$$

Next, let r denote the assertion **false** or $\bar{x} = \bar{y}$. We have that

$$\models_{sp} \{WP_{sp}(P_i(\bar{t}_i)), r\}\ P_i(\bar{t}_i)\ \{r\}$$

implies

$$\models_{sp} \{WP_{sp}(P_i(\bar{t}_i), r)\}\ \textbf{begin local}\ \bar{u}_i := \bar{t}_i;\ S_i\ \textbf{end}\ \{r\},$$

for every $P_i(\bar{u}_i) :: S_i \in D$. By Lemma 9.8 again we have

$$A \vdash_{SPR} \{WP_{sp}(P_i(\bar{t}_i), r)\}\ \textbf{begin local}\ \bar{u}_i := \bar{t}_i;\ S_i\ \textbf{end}\ \{r\},$$

for every assumption of A. Finally, by the recursion III rule 15, we conclude that $\{p\}\ S\ \{q\}$ is derivable in the proof system SPR^+. □

10. Extensions

The approach to the verification of the object-oriented programs that we proposed here is flexible and natural. To substantiate this claim we explain now how it can be naturally extended to other features of object-oriented programming and to total correctness.

10.1. Access to instance variables

A natural possibility is to allow method calls to access instance variables of arbitrary objects, a feature available in Java. Then, given instance variables x, y and object variables s, t, we could use assignments such as $y := s.x + 1$, or $s.x := t.y + 1$, and use global expressions in Boolean expressions, for example $2 \cdot s.x = t.y + 1$, and as actual parameters in method calls, for example $s.m(t.y + 1)$.

To extend the obtained results to the resulting programming language the presentation would have to be modified in a number of places. More precisely, such an extension requires the following:

- introduction of the global expressions already in Section 3.1,
- introduction of *global terms*, which are expressions built out of global expressions using the admitted function symbols (and respecting the well-typedness condition),
- extension of the assignment statement to one of the form $s := t$, where s is a global expression and t is a global term,
- admission of the method calls of the form $s.m(t_1, \ldots, t_n)$, where s is an object expression and $t_1, \ldots, t_n$ are global terms,
- introduction of the definition of semantics of global terms in Section 4.1,
- extension of the notion of an update of a state $\sigma[s := d]$ in Section 4.2 to the case of a global expression s,
- extension of the definition of substitution given in Section 6.2 to one of the form $[s := t]$, where s is a global expression and t is a global term,
- extension of the transformation Θ given in Section 5 to the considered programming language, by defining $\Theta(s)$ for a global expression s already in Section 5.2,
- extension of the assignment axiom 11 to the above introduced class of assignments, and the recursion rules 13 and 14 to the above introduced method calls,
- extension of the results of Section 8, notably the Homomorphism Lemma 8.8, to this extended programming language.

The details are relatively straightforward and omitted.

10.2. Object creation

Most existing approaches to object creation (see for example in [7]) follow implicitly the transformational approach by modeling it in terms of object *activation*. This can be made more explicit as follows. Given an array variable *store* of type $\mathbb{N} \to$ **object** and a variable *count* of type $\mathbb{N}$, we model the object creation statement $x :=$ **new** by the statement

$$count := count + 1;\ x := store[count].$$

This modeling of the object activation crucially depends on *store* being an *unbounded* array variable. Further it assumes that *store* is injective:

$$\forall i : \forall j : i \neq j \rightarrow store[i] \neq store[j],$$

which is possible since we assumed that the type **object** has infinitely many elements.

A drawback of this approach to object creation is that it involves an explicit reference to a particular implementation. Since object variables can only be compared for equality or dereferenced, we show in Chapter 6 of our book [3] that we can in fact define a *substitution* $[x := \mathbf{new}]$ which statically evaluates expressions in which x occurs, assuming that x denotes a newly created object. This in turn allows us to define the weakest precondition $p[x := \mathbf{new}]$ of the object creation statement $x := \mathbf{new}$ and w.r.t. a postcondition p which abstracts from the particular implementation of object creation. This yields the assignment statement

$$\big\{p[x := \mathbf{new}]\big\}\ x := \mathbf{new}\ \{p\}$$

allowing us to reason about object creation.

10.3. Classes and inheritance

The transformational approach discussed in this paper can be readily extended to deal with various features of mainstream object-oriented languages, like classes, inheritance and polymorphism (i.e., subtyping). As an example, we now discuss the details of such a transformation for a fragment of Java that extends the object-oriented language considered so far with *dynamic binding of methods*. This extension comprises the following:

- introduction of *classes* as basic types,
- use within the context of each program of a reflexive and transitive *subclass relation* and its inverse *superclass relation* defined on the set of classes used; we assume that this relation respects *single inheritance*, i.e., each class has at most one direct superclass, and that **object** is the superclass of each class,
- introduction of the assignment $u := t$, where the type of the object expression t is a subclass of the type of u, and of the method call $s.m(t_1, \ldots, t_n)$, where for $i \in \{1, \ldots, n\}$ the type of the actual parameter t_i is a subclass of the type of the corresponding formal parameter, in case t_i is an object expression,
- introduction of mutually disjoint sets $\mathcal{D}_C \subseteq \mathcal{D}_{\mathbf{object}}$ of object instances of class C.

Further, we associate with each class a set of method declarations. The instance variables of a class C are the inherited ones plus the ones that are introduced in the method declarations of C. An object-oriented program in this new setting consists then of a main statement and a set of classes, each with its set of method declarations, and a subclass relation. The semantics of a method call in this extension is captured by the rule

$$\big\langle s.m(\bar{t}), \sigma \big\rangle \rightarrow \langle \mathbf{if}\ s \neq \mathbf{null} \rightarrow \mathbf{begin\ local\ this}, \bar{u} := s, \bar{t};\ S\ \mathbf{fi\ end}, \sigma \rangle,$$

where S is such that $\sigma(s) \in \mathcal{D}_C$ and $m(\bar{u}) :: S \in C$. In words, the class of the object denoted by the expression s determines the actual definition of the called method to be used. Note that this class in general is a subclass of the type of the expression s.

We now explain how the programs formed in this extended setting can be transformed to the programs considered earlier extended by an introduction for each class C of a unary predicate $C : \mathbf{object} \rightarrow \mathbf{Boolean}$ whose semantics is defined by

$$\sigma\big(C(s)\big) = \begin{cases} \mathbf{true} & \text{if } \sigma(s) \in \mathcal{D}_C, \\ \mathbf{false} & \text{otherwise.} \end{cases}$$

In order to model dynamic binding in this extended object-oriented language we first *flatten* the inheritance hierarchy between classes by introducing a global set of method definitions D which consists of all method definitions

$$m@C(\bar{u}) :: S,$$

where the declaration $m(\bar{u}) :: S$ appears in the class C itself or in the 'minimal' superclass C' of C, that is, no other superclass of C which is also a subclass of C' contains a declaration of m. We then model the semantics of a method call $s.m(\bar{t})$ by the statement S_n, inductively defined for $i \in \{0, \ldots, n-1\}$ by

$$S_0 \equiv skip,$$

$$S_{i+1} \equiv \mathbf{if}\ C_{i+1}(s)\ \mathbf{then}\ s.m@C_{i+1}(\bar{t})\ \mathbf{else}\ S_i\ \mathbf{fi},$$

where $\{C_1, \ldots, C_n\}$ is the set of subclasses of the type of s.

After establishing an analogue of Theorem 5.4 for the above transformation one could verify the programs written in the source language by verifying their translated version. In principle one could also derive proof rules that deal with the source programs directly, analogously as in Section 7.

10.4. Total correctness

To focus on the crucial aspects of our approach to verification we did not deal with program termination. The appropriate extension combines strong partial correctness with termination and requires the following:

- addition of a special state $\perp$ that models divergence,
- modification of the definition of semantics to take care of divergence,
- introduction of a new notion of soundness of a proof system,
- replacement of the current LOOP rule 6 by a rule that also takes care of termination,
- replacement of the current the recursion rules 13 and 14 by a single rule that also takes care of termination,
- similarly for the recursion III rule 15,
- appropriate modification of the proofs in Section 8 to additionally deal with termination.

The details are presented in [3, Chapter 6]. Since termination is, roughly speaking, orthogonal to object-orientation, the transformational approach for (strong) partial correctness can be extended to total correctness in a straightforward, though somewhat tedious, manner.

11. Conclusion

We presented here an assertional proof system to reason about partial and strong partial correctness of a class of object-oriented programs. Its formal justification (that is, soundness and relative completeness) was carried out using a syntax-directed transformation to recursive programs.

We proved a new relative completeness result for a class of recursive programs that use variables ranging over abstract data types (like the basic type **object**) and showed that the transformation preserves completeness. We also showed that the transformational approach can be applied to intricate and complex object-oriented features, such as inheritance and subtype polymorphism, by transforming them in the context of a *closed* program to the core language considered in this paper.

Extension of the transformational approach to *open* object-oriented programs, so programs that do not necessarily include the definitions of all the classes used (in Java for example such classes are imported from *packages*), however, requires an additional study of structuring recursive programs by means of *modules* along the lines of the Modula programming language [31] and of the corresponding proof-theoretical concept of a *contract* as introduced in the Eiffel programming language [20].

Appendix A. Semantics

In the following we list the omitted transition axioms and rules that define the transition relation $\rightarrow$.

(i) $\langle skip, \sigma\rangle \rightarrow \langle E, \sigma\rangle$,
(ii) $\langle u := t, \sigma\rangle \rightarrow \langle E, \sigma[u := \sigma(t)]\rangle$, where $u \in Var$ is a simple variable or $u \equiv a[s_1, \ldots, s_n]$, for $a \in Var$,
(iii) $\langle \bar{x} := \bar{t}, \sigma\rangle \rightarrow \langle E, \sigma[\bar{x} := \sigma(\bar{t})]\rangle$,
(iv) $$\frac{\langle S_1, \sigma\rangle \rightarrow \langle S_2, \tau\rangle}{\langle S_1;\ S, \sigma\rangle \rightarrow \langle S_2;\ S, \tau\rangle},$$
(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \rightarrow \langle S_1, \sigma\rangle$, where $\sigma \models B$,
(vi) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \rightarrow \langle S_2, \sigma\rangle$, where $\sigma \models \neg B$,
(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma\rangle \rightarrow \langle S;\ \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma\rangle$, where $\sigma \models B$,
(viii) $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma\rangle \rightarrow \langle E, \sigma\rangle$ where $\sigma \models \neg B$.

Appendix B. Axioms and proof rules

In the following we list the used axioms and proof rules. Given an assertion q we denote below its set of free variables by $free(q)$.

B.1. Axioms and proof rules for the kernel language

To establish correctness of programs from the kernel language of Section 2 we rely on the following axioms and proof rules.

AXIOM 1. SKIP

$$\{p\}\ skip\ \{p\}.$$

AXIOM 2. ASSIGNMENT

$$\big\{p[u := t]\big\}\ u := t\ \{p\}$$

where $u \in Var$ or $u \equiv a[s_1, \ldots, s_n]$ and $a \in Var$.

AXIOM 3. PARALLEL ASSIGNMENT

$$\big\{p[\bar{x} := \bar{t}]\big\}\ \bar{x} := \bar{t}\ \{p\}.$$

RULE 4. COMPOSITION

$$\frac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}.$$

RULE 5. CONDITIONAL

$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}.$$

RULE 6. LOOP

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg B\}}.$$

RULE 7. CONSEQUENCE

$$\frac{p \to p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \to q}{\{p\}\ S\ \{q\}}.$$

RULE 8. FAILURE

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \textbf{if}\ B \to S\ \textbf{fi}\ \{q\}}.$$

RULE 9. FAILURE II

$$\frac{p \to B,\ \{p\}\ S\ \{q\}}{\{p\}\ \textbf{if}\ B \to S\ \textbf{fi}\ \{q\}}.$$

RULE 10. BLOCK

$$\frac{\{p\}\ \bar{x} := \bar{t};\ S\ \{q\}}{\{p\}\ \textbf{begin local}\ \bar{x} := \bar{t};\ S\ \textbf{end}\ \{q\}}$$

where $\{\bar{x}\} \cap free(q) = \emptyset$.

B.2. Auxiliary rules

Further, we rely on the following auxiliary axioms and proof rules that occasionally refer to the assumed set of procedure or method declarations D. We refer in them to the sets of variables $var(D)$ and $change(D)$ defined in the expected way.

RULE A1. DISJUNCTION

$$\frac{\{p\}\ S\ \{q\},\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}.$$

RULE A2. CONJUNCTION

$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}.$$

RULE A3. $\exists$-INTRODUCTION

$$\frac{\{p\}\ S\ \{q\}}{\{\exists x : p\}\ S\ \{q\}},$$

where $x \notin var(D) \cup var(S) \cup free(q)$.

RULE A4. INVARIANCE

$$\frac{\{r\}\ S\ \{q\}}{\{p \wedge r\}\ S\ \{p \wedge q\}}$$

where $free(p) \cap (change(D) \cup change(S)) = \emptyset$.

RULE A5. SUBSTITUTION

$$\frac{\{p\}\ S\ \{q\}}{\{p[\bar{z} := \bar{t}]\}\ S\ \{q[\bar{z} := \bar{t}]\}}$$

where $var(\bar{z}) \cap (var(D) \cup var(S)) = var(\bar{t}) \cap (change(D) \cup change(S)) = \emptyset$.

B.3. Axioms and proof rules for object-oriented programs

The following axioms and proof rules were introduced for the object-oriented programs.

AXIOM 11. ASSIGNMENT TO INSTANCE VARIABLES

$$\big\{p[u := t]\big\}\ u := t\ \{p\}$$

where u is a simple or subscripted instance variable.

RULE 12. WEAKENING

$$\frac{\{p \wedge s \neq \textbf{null}\}\ s.m(\bar{t})\ \{q\}}{\{p\}\ s.m(\bar{t})\ \{q\}}$$

RULE 13. RECURSION I

$$\frac{\begin{array}{l}\{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash \{p\}\ S\ \{q\},\\ \{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash\\ \quad \{p_i\}\ \textbf{begin local this}, \bar{u}_i := s_i, \bar{t}_i;\ S_i\ \textbf{end}\ \{q_i\},\ i \in \{1, \ldots, n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $m_i(\bar{u}_i) :: S_i \in D$ for $i \in \{1, \ldots, n\}$.

RULE 14. RECURSION II

$$\frac{\begin{array}{l}\{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash \{p\}\ S\ \{q\},\\ \{p_1\}\ s_1.m_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ s_n.m_n(\bar{t}_n)\ \{q_n\} \vdash\\ \quad \{p_i\}\ \textbf{begin local this}, \bar{u}_i := s_i, \bar{t}_i;\ S_i\ \textbf{end}\ \{q_i\},\ i \in \{1, \ldots, n\}\\ p_i \to s_i \neq \textbf{null},\ i \in \{1, \ldots, n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $m_i(\bar{u}_i) :: S_i \in D$ for $i \in \{1, \ldots, n\}$.

B.4. Proof rule for recursive programs

Finally, the following proof rule was introduced for the recursive programs.

RULE 15. RECURSION III

$$\frac{\begin{array}{l}\{p_1\}\ P_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{t}_n)\ \{q_n\} \vdash \{p\}\ S\ \{q\},\\ \{p_1\}\ P_1(\bar{t}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{t}_n)\ \{q_n\} \vdash\\ \quad \{p_i\}\ \textbf{begin local}\ \bar{u}_i := \bar{t}_i;\ S_i\ \textbf{end}\ \{q_i\},\ i \in \{1, \ldots, n\}\end{array}}{\{p\}\ S\ \{q\}}$$

where $P_i(\bar{u}_i) :: S_i \in D$.

Appendix C. Proof systems

In the following we list the proof systems used in this paper.

Kernel language
PROOF SYSTEM *PK* for partial correctness:

This system consists of the group of axioms and rules 1–8, and 10.

PROOF SYSTEM *SPK* for strong partial correctness:

This system consists of the group of axioms and rules 1–7, 9, and 10.

Object-oriented programs
PROOF SYSTEM *PO* for partial correctness:

This system is obtained by extending *PK* with the axiom 11 for assignments to instance variables, the weakening rule 12, and the auxiliary rules A1–A5.

PROOF SYSTEM PO^+ for strong partial correctness:

This system is obtained by extending *PO* by the recursion I rule 13.

PROOF SYSTEM *SPO* for partial correctness:

This system is obtained by extending *SPK* with the axiom 11 for assignments to instance variables and the auxiliary rules A1–A5.

PROOF SYSTEM SPO^+ for strong partial correctness:

This system is obtained by extending *SPR* by the recursion II rule 14.

Recursive programs
PROOF SYSTEM *PR* for partial correctness:

This system is obtained by extending *PK* with the auxiliary rules A1–A5.

PROOF SYSTEM *SPR* for strong partial correctness:

This system is obtained by extending *SPK* with the auxiliary rules A1–A5.

PROOF SYSTEM PR^+ for partial correctness:

This system is obtained by extending *PR* by the recursion III rule 15.

PROOF SYSTEM SPR^+ for strong partial correctness:

This system is obtained by extending the *SPR* by the recursion III rule 15.

References

[1] M. Abadi, K. Leino, A logic of object-oriented programs, in: N. Dershowitz (Ed.), Verification: Theory and Practice, in: Lecture Notes in Comput. Sci., vol. 2772, Springer, 2003, pp. 11–41.
[2] P. America, F.S. de Boer, Proving total correctness of recursive procedures, Inform. and Comput. 84 (2) (1990) 129–162.
[3] K.R. Apt, F.S. de Boer, E.-R. Olderog, Verification of Sequential and Concurrent Programs, third, extended edition, Springer, London, 2009.
[4] K.R. Apt, N. Francez, W.P. de Roever, A proof system for communicating sequential processes, ACM Trans. Program. Lang. Syst. 2 (3) (1980) 359–385.
[5] A. Banerjee, D.A. Naumann, Ownership confinement ensures representation independence for object-oriented programs, J. ACM 52 (6) (2005) 894–960.
[6] M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, K.R.M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: Formal Methods for Components and Objects (FMCO), 2005, pp. 364–387.
[7] B. Beckert, R. Hähnle, P.H. Schmitt (Eds.), Verification of Object-Oriented Software: The KeY Approach, Lecture Notes in Comput. Sci., vol. 4334, Springer, 2007.
[8] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, E. Poll, An overview of JML tools and applications, Int. J. Softw. Tools Technol. Transf. 7 (3) (2005) 212–232.
[9] E.M. Clarke, Programming language constructs for which it is impossible to obtain good Hoare axiom systems, J. ACM 26 (1) (January 1979) 129–147.
[10] S.A. Cook, Soundness and completeness of an axiom system for program verification, SIAM J. Comput. 7 (1) (1978) 70–90.
[11] J.W. de Bakker, Mathematical Theory of Program Correctness, Prentice Hall International, Englewood Cliffs, NJ, 1980.
[12] F. S. de Boer, Reasoning about dynamically evolving process structures – A proof theory of the parallel object-oriented language POOL, PhD thesis, Free University of Amsterdam, 1991.

[13] G. Gorelick, A complete axiomatic system for proving assertions about recursive and non recursive programs, Technical Report 75, Department of Computer Science, University of Toronto, 1975.
[14] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580, 583.
[15] C.A.R. Hoare, Procedures and parameters: an axiomatic approach, in: E. Engeler (Ed.), Proceedings of Symposium on the Semantics of Algorithmic Languages, in: Lecture Notes in Math., vol. 188, Springer, 1971, pp. 102–116.
[16] M. Huisman, B. Jacobs, Java program verification via a Hoare logic with abrupt termination, in: T.S.E. Maibaum (Ed.), Fundamental Approaches of Software Engineering (FASE), in: Lecture Notes in Comput. Sci., vol. 1783, Springer, 2000, pp. 284–303.
[17] B. Jacobs, Weakest pre-condition reasoning for Java programs with JML annotations, J. Log. Algebr. Program. 58 (1–2) (2004) 61–88.
[18] G. Klein, T. Nipkow, A machine-checked model for a Java-like language, virtual machine, and compiler, ACM Trans. Program. Lang. Syst. 28 (4) (2006) 619–695.
[19] H. Langmaack, E.-R. Olderog, Present-day Hoare-like systems for programming languages with procedures: Power, limits and most likely expressions, in: Proc. 7th Intern. Colloq. on Automata, Languages and Programming (ICALP), 1980, pp. 363–373.
[20] B. Meyer, Object-Oriented Software Construction, second ed., Prentice Hall, 1997.
[21] J.M. Morris, A general axiom of assignment/Assignment and linked data structures/A proof of the Schorr–Waite algorithm, in: Theoretical Foundations of Programming Methodology, in: Lecture Notes of an International Summer School, Reidel, 1982.
[22] P. Müller, A. Poetzsch-Heffter, G.T. Leavens, Modular invariants for layered object structures, Sci. Comput. Programming 62 (3) (2006) 253–286.
[23] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Inform. 6 (1976) 319–340.
[24] S. Owicki, D. Gries, Verifying properties of parallel programs: an axiomatic approach, Commun. ACM 19 (1976) 279–285.
[25] C. Pierik, F.S. de Boer, A proof outline logic for object-oriented programming, Theoret. Comput. Sci. 343 (3) (2005) 413–442.
[26] G.D. Plotkin, A structural approach to operational semantics, Technical Report DAIMI-FN 19, Department of Computer Science, Aarhus University, 1981.
[27] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60–61 (2004) 17–139, revised version of Plotkin [26].
[28] J.C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Logic in Computer Science (LICS), 2002, pp. 55–74.
[29] J.V. Tucker, J.I. Zucker, Program Correctness over Abstract Data Types, with Error-State Semantics, CWI Monograph, North-Holland, Amsterdam, 1988.
[30] D. von Oheimb, Hoare logic for Java in Isabelle/HOL, Concurr. Comput. Practice Experience 13 (13) (2001) 1173–1214.
[31] N. Wirth, Programming in Modula-2, Texts Monogr. Comput. Sci., Springer-Verlag, ISBN 0-387-50150-9, 1989.