Verification of Object-Oriented Programs: a Transformational Approach

Krzysztof R. Apt\textsuperscript{1,2}, Frank S. de Boer\textsuperscript{1,3}, and Ernst-Rüdiger Olderog\textsuperscript{4}

\textsuperscript{1}Centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands
\textsuperscript{2}University of Amsterdam, Institute of Language, Logic and Computation, Amsterdam
\textsuperscript{3}Leiden Institute of Advanced Computer Science, University of Leiden, The Netherlands
\textsuperscript{4}Department of Computing Science, University of Oldenburg, Germany

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.

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 \texttt{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 [22] and [23]), 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 \textit{encapsulate}) their own local variables, and
- objects interact via \textit{method} calls.

The execution of a method call involves a temporary \textit{transfer} of control from the local state of the caller object to that of the called object (also referred to by \textit{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 the preservation of soundness. The aim of this paper is to provide a systematic and self-contained presentation which focuses on (relative) completeness. Readers interested in example correctness proofs in the proposed proof systems may consult pages 226–237 of [3].

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 [24] 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 pre-condition 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 [20]. A more recent approach is that of separation logic described in [27]. 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 [21] 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 [29] 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 [28] 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 language features and with total correctness. In the appendix 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. Further on, for each \( n \geq 1 \) we consider the following higher types:

\[
T_1 \times \ldots \times T_n \rightarrow T,
\]

where \( T_1, \ldots, T_n, T \) are basic types. Here \( T_1, \ldots, T_n \) are called argument types and \( T \) the value type. A type should be viewed as a name, or a notation for the intended set of values. The type integer denotes the set of all integers, the type Boolean the set \{true, false\}, and, as usual, a type \( T_1 \times \ldots \times T_n \rightarrow T \) the set of all functions from the Cartesian product of the sets denoted by \( T_1, \ldots, T_n \) to the set denoted by \( T \).

We use simple variables of a basic type and array variables of a higher type. By Var we denote the set of variable declarations. In general, however, we omit the typing information and identify a variable declaration with the variable name, that is, we drop the type information. In each example it will be clear what variable declarations are assumed.

Out of typed variables and typed constants typed expressions are constructed, in particular integer and Boolean expressions. To deal later with aliasing we allow conditional expressions of the form if \( B \) then \( t_1 \) else \( t_2 \) fi. A subscripted variable derived from an array variable \( a \) of type \( T_1 \times \ldots \times T_n \rightarrow T \) is an expression of the form \( a[t_1, \ldots, t_n] \), where each expression \( t_i \) is of type \( T_i \).

In this section we introduce a small kernel programming language. We start by taking programs generated by the following grammar:

\[
S ::= \text{skip} \mid u := t \mid \vec{x} := \vec{t} \mid S_1; S_2 \mid \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } B \text{ do } S_1 \text{ od,}
\]
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 modelling of the parameter passing. As an abbreviation we introduce $\text{if } B \text{ then } S \text{ fi } \equiv \text{if } B \text{ then } S \text{ else skip fi }$.

Given an expression $t$, we denote by $\text{var}(t)$ the set of all simple and array variables that appear in $t$. Analogously, given a program $S$, we denote by $\text{var}(S)$ the set of all simple and array variables that appear in $S$, and by $\text{change}(S)$ the set of all 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.

Next, to check certain conditions during the execution we introduce the following failure statement:

$$S ::= \text{if } B \rightarrow S_1 \text{ fi }$$

where $B$ is a Boolean expression (representing the desired condition) and $S_1$ is a statement. The statement $\text{if } B \rightarrow S_1 \text{ fi }$ is executed as follows. First $B$ is evaluated. If it is true, then $S_1$ is executed; otherwise a failure arises. This differs from the execution of the conditional statement $\text{if } B \text{ then } S_1 \text{ fi }$, which terminates if $B$ evaluates to false.

Further, to distinguish between local and global variables, we introduce a block statement:

$$S ::= \text{begin local } \bar{x} := \bar{t}; S_1 \text{ end }.$$

A block statement introduces a non-empty sequence $\bar{x}$ of simple distinct local variables, all of which are explicitly initialized by means of a parallel assignment $\bar{x} := \bar{t}$, and provides an explicit scope for these simple local variables.

We shall see that the definition of the semantics of the block statement treats correctly nested blocks. Also, it interprets the occurrences of the variables from $\bar{x}$ in $\text{var}(\bar{t})$ as global variables (w.r.t. the considered block statement). This extends the customary use of initializations of local variables in a block statement in a way that allows us to handle correctly the call-by-value mechanism without renaming program variables, both in the definition of semantics and proof theory.

### 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 [26]. As usual, it is defined in terms of transitions between configurations. A configuration $C$ is a pair $< S, \sigma >$ consisting of a statement $S$ that is to be executed and a state $\sigma$, i.e., a mapping that assigns to each variable (including local variables) of type $T$ a value drawn from the set $D_T$ denoted by type $T$.

Given a state $\sigma$ and an expression $t$, we define in a standard way its semantics $\sigma(t)$, the value assigned to it by $\sigma$. 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 $\sigma$. 


We denote the set of states by $\Sigma$. Unless stated otherwise, the letters $\sigma, \tau$ range over $\Sigma$. We use a special state $\text{fail}$ to represent an abnormal situation in a program execution, a failure in an execution of a program. We stipulate that $\text{fail} \not\in \Sigma$. Sometimes to avoid confusion we refer to the elements of $\Sigma$ as proper states.

For the semantics of (parallel) assignments we need the notion of a state update of a proper state $\sigma$, 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 $\sigma$ 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 $\sigma$ except for the variable $a$ where the value $\sigma(a)(\sigma(t_1), \ldots, \sigma(t_n))$ is changed to $d$. We extend the definition of update to the special state by putting $\text{fail}[u := d] = \text{fail}$. 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 $\sigma$ except for $u_i$ where its value is $d_i$, for $i \in \{1, \ldots, n\}$.

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

- $< \text{if } B \rightarrow S \text{ fi, } \sigma > \rightarrow < S, \sigma >$, where $\sigma \models B$,
- $< \text{if } B \rightarrow S \text{ fi, } \sigma > \rightarrow < E, \text{ fail } >$, where $\sigma \models \neg B$,
- $< \text{begin local } \bar{x} := \bar{t}; S \text{ end, } \sigma > \rightarrow < \bar{x} := \bar{t}; S; \bar{x} := \sigma(\bar{x}), \sigma >$.

The second axiom explains the need for the use of the special state $\text{fail}$. The last axiom ensures that the local variables are initialized as prescribed by the parallel assignment and that upon termination the global variables whose names coincide with 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 and are listed in Appendix A. Based on the transition relation $\rightarrow$ we consider two variants of input/output semantics for a program $S$. The partial correctness semantics is a mapping $M[S] : \Sigma \rightarrow \mathcal{P}(\Sigma)$ defined by

$$M[S](\sigma) = \{ \tau \in \Sigma \mid < S, \sigma > \rightarrow^* < E, \tau > \},$$

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

$$M_{sp}[S](\sigma) = \{ \tau \in \Sigma \cup \{\text{fail}\} \mid < S, \sigma > \rightarrow^* < E, \tau > \}.$$ 

So for all $S$ and $\sigma$ we have $\text{fail} \not\in M[S](\sigma)$, while for some $S$ and $\sigma$ we can have $\text{fail} \in M_{sp}[S](\sigma)$. In the latter case we say that $S$ can fail when started in $\sigma$.

We extend these semantic mappings to sets of states by defining for $X \subseteq \Sigma$ and $\mathcal{N}$ standing for $M$ or $M_{sp}$:

$$\mathcal{N}[S](X) = \bigcup_{\sigma \in X} \mathcal{N}[S](\sigma).$$
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 $D_{\text{object}}$.

3.1 Expressions

An expression (e.g., a variable) of type object denotes an object. Simple variables of type object and array variables with value type object are called object variables. As a special case, we distinguish the simple object variable this which in each state denotes the currently executing object.

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

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, this $\in \text{Var}$.

The only operation of a higher type which involves the basic type object (as argument type or as value type) is the equality $=_{\text{object}}$ (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.

Example 1. Given an array variable $a \in \text{Var} \cup \text{IVar}$ of type integer $\rightarrow$ object,

- $a[0]$ is an expression of type object,
- $\text{this} = a[0]$ and $\text{this} \neq \text{null}$ are Boolean expressions.

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 first extend the syntax of the kernel language introduced in Section 2 by the assignment to instance variables:

$$ S ::= u ::= t, $$
where $u \in IVar$ is an instance variable. The syntax of method calls is described by the following clause:

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

where $n \geq 0$. 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 only consider statements in which this is read-only, that is, we disallow assignments to the variable this. Further, to ensure that instance variables are permanent, we require that in each block statement $\text{begin local } \bar{u} := \bar{t}; S \text{ end}$ we have $\text{var}(\bar{u}) \subseteq \text{Var} \setminus \{\text{this}\}$, that is, instance variables are not used as local variables. 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).

Apart from denoting the callee of a method call, object expressions, as already indicated in Example 1, can appear in Boolean expressions. Further, we allow for assignments to object variables. In particular, the assignment $u := \text{null}$ is used to assign to the object variable $u$ the void reference. In turn, the assignment $u := v$ for the object variables $u$ and $v$ causes $u$ and $v$ to point to the same object.

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 $D$. Additionally, we assume that the method calls are well-typed, which means that the numbers of formal and actual parameters agree and that for each parameter position the types of the corresponding actual and formal parameters coincide. Finally, we assume that the name clashes between local variables and global variables are resolved by assuming that no local variable of $S$ occurs in $D$. If $D$ is clear from the context we refer to the main statement as an object-oriented program.

**Example 2.** Consider the object-oriented program

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

in the context of the recursive method declaration

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

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.

\[ \square \]

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 (‘in-
stance’) in each object. Conceptually, when defining the semantics of an instance vari-
able u we view it as a variable of the form this.u, where this represents the current
object. So, given a proper state \( \sigma \) and a simple instance variable \( x \) we first determine
the current object \( o \), which is \( \sigma(\text{this}) \). Then we determine the local state of this object,
which is \( \sigma(o) \), or \( \sigma(\sigma(\text{this})) \), and subsequently apply this local state to the
considered instance variable \( x \). This means that given a proper state \( \sigma \) the value assigned to
the instance variable \( x \) is \( \sigma(o)(x) \), or, written out in full, \( \sigma(\sigma(\text{this}))(x) \). This two-step
procedure is at the heart of the definition of semantics of an expression given below.

Next, we introduce a value null \( \in D_{\text{object}} \). So in each proper state each variable
of type object equals some object of \( D_{\text{object}} \), which can be the null object. A proper
state \( \sigma \) now additionally assigns to each object \( o \in D_{\text{object}} \) 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 vari-
able. Note that by definition a proper state also assigns to null a local state. However,
by Lemma 1 from Subsection 4.3 below this state is not reachable in any computation.

So the value of an instance variable \( x \in IVar \) of an object \( o \in D_{\text{object}} \) is given by
\( \sigma(o)(x) \) and, as before, the value of a normal variable \( x \in Var \) is given by \( \sigma(x) \).

Note that the local state of the current object \( \sigma(\text{this}) \) is given by \( \sigma(\sigma(\text{this})) \). Fur-
ther, note that in particular, if an instance variable \( x \) is of type Object, then for each
object \( o \in D_{\text{object}} \), \( \sigma(o)(x) \) is either null or an object \( o' \in D_{\text{object}} \), whose local
state is \( \sigma(o') \), i.e., \( \sigma(\sigma(o)(x)) \). This application of \( \sigma \) can of course be nested, to get
local states of the form \( \sigma(\sigma(\sigma(o')(x))) \), etc. Note that a state \( \sigma \) also assigns a local state
\( \sigma(\text{null}) \) to the null object. However, as we will see below this local state will ne-
ver be accessed by an object-oriented program.

The additional component which assigns to each object its local state provides a
mathematical representation of the heap. The heap allocates to each object a region
of the memory that stores the values of its instance variables. In Figure 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.

Formally, we extend the semantics \( \sigma(s) \) of an expression \( s \) in a proper state \( \sigma \)
(cf. Subsection 2.2) by the following clauses:
4.2 Updates of states

Next, we proceed with the revision of the definition of a state update for the case of instance variables. Here, the intuition we provided when explaining the semantics of instance variables, is of help. Consider a proper state \( \sigma \), a simple instance variable \( x \), and a value \( d \) belonging to the type of \( x \). To perform the corresponding state update...
of $\sigma$ on $x$ to $d$, written as $\sigma[x := d]$, we first identify the current object $o$, which is $\sigma(\texttt{this})$ and its local state, which is $\sigma(o)$, or $\sigma(\sigma(\texttt{this}))$, that we denote by $\tau$. Then we perform the appropriate update on the state $\tau$. So the desired update of $\sigma$ is achieved by modifying $\tau$ to $\tau[x := d]$.

In general, let $u$ be a (possibly subscripted) instance variable of type $T$ and $\tau$ a local state. We define for $d \in D_T$ $\tau[u] := d$ analogously to the standard definition of state update for normal variables. Furthermore, we define for an object $o \in D_{\text{object}}$ and local state $\tau$, the state update $\sigma[o := \tau]$ by

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

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 $d \in D_T$, as follows:

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

where $o = \sigma(\texttt{this})$ 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 $\tau$. In its fully expanded form we get the following difficult to parse definition of a state update:

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

Example 3. Let $x$ be a Boolean instance variable, $o = \sigma(\texttt{this})$, and $\tau = \sigma(o)$. Then

$$
\sigma[x := \texttt{true}](x) = \{ \text{(1) with $\sigma$ replaced by $\sigma[x := \texttt{true}]$} \}
\sigma[x := \texttt{true}](\sigma[x := \texttt{true}](\texttt{this}))(x) = \{ \text{by the definition of state update, $\sigma[x := \texttt{true}](\texttt{this}) = \sigma(\texttt{this}) = o$} \}
\sigma[x := \texttt{true}](o)(x) = \{ \text{definition of state update $\sigma[x := \texttt{true}]$} \}
\sigma[o := \tau[x := \texttt{true}]](o)(x) = \{ \text{definition of state update $\sigma[o := \tau][x := \texttt{true}]$} \}
\tau[x := \texttt{true}](x) = \{ \text{definition of state update $\tau[x := \texttt{true}]$} \}
\texttt{true}.
\]

\[\square\]

4.3 Semantics of programs

For the operational semantics of the considered programs we introduce three 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.
\[ < u := t, \sigma > \rightarrow < E, \sigma[u := \sigma(t)] >, \]
\[ < s.m(\bar{t}), \sigma > \rightarrow < \text{begin local this, } \bar{u} := s, \bar{t}; S \text{ end, } \sigma >, \]
\[ \text{where } \sigma(s) \neq \text{null and } m(\bar{u}) :: S \in D, \]
\[ < s.m(\bar{t}), \sigma > \rightarrow < E, \text{fail} >, \text{ where } \sigma(s) = \text{null}. \]

Thanks to the extended definition of a state update, the first transition axiom models the meaning of an assignment to an instance variable in exactly the same way as the meaning of the assignment to a normal variable. By ‘unfolding’ the right-hand side of the second transition axiom, using the transition axioms for the block and parallel assignment statements, we get
\[ < s.m(\bar{t}), \sigma > \rightarrow^* < S; \text{this, } \bar{u} := \sigma(\text{this, } \bar{u}), \sigma[\text{this, } \bar{u} := \sigma(s, \bar{t})] >, \]

where \( \sigma(s) \neq \text{null} \) and \( m(\bar{u}) :: S \in D \).

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 \( \text{this} \)) becomes \( \sigma(s) \), and upon termination of the method body \( S \) the current object is restored to its previous value \( \sigma(\text{this}) \) using the parallel assignment \( \sigma[\text{this, } \bar{u} := \sigma(s, \bar{t})] \).

The third transition axiom shows that if in the considered state \( \sigma \) the called object \( s \) equals the void reference (it equals \( \text{null} \)), then the method call yields a failure.

**Lemma 1.** (Safety) For every statement \( S \) that can arise during an execution of an object-oriented program and every proper state \( \sigma \), the following holds.

(i) **Absence of null Reference:** if \( \sigma(\text{this}) \neq \text{null} \) and \( < S, \sigma > \rightarrow < S_1, \tau >, \)
then \( \tau(\text{this}) \neq \text{null} \).

(ii) **Type Safety:** if \( S \) is well-typed and \( < S, \sigma > \rightarrow < S_1, \tau > \) holds, then also \( S_1 \) is well-typed.

**Proof.** (i) If \( S \neq E \) then any configuration \( < S, \sigma > \) has a successor in the transition relation \( \rightarrow \). To prove the preservation of the assumed property of the state it suffices to consider the execution of an assignment \( \text{this} := s \). Each such assignment arises only within the context of the block statement in the corresponding transition axiom and is activated in a state \( \sigma \) such that \( \sigma(s) \neq \text{null} \). This yields a state \( \tau \) such that \( \tau(\text{this}) \neq \text{null} \).

(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. \( \square \)

When considering verification of object-oriented programs we shall only consider computations that start in a proper state \( \sigma \) such that \( \sigma(\text{this}) \neq \text{null} \), i.e., in a state in which the current object differs from the void reference. The Safety Lemma 1 implies that such computations never lead to a proper state in which this inequality is violated.

The partial correctness semantics \( M[S] \) and the strong partial correctness semantics \( M_{sp}[S] \) 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 \( n \geq 0 \), 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. So we allow mutually recursive procedures but not nested procedures. When considering recursive programs we assume that all procedures whose calls appear in the considered recursive programs are declared in \( D \). Additionally, we assume that the procedure calls are well-typed in the same sense as method calls. Finally, we assume that the name clashes between local variables and global variables are resolved by assuming that no local variable of \( S \) occurs in \( D \).

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.

\[ \langle P(\bar{t}), \sigma \rangle \rightarrow \langle \text{begin local } \bar{u} := \bar{t}; S \text{ end}, \sigma \rangle, \text{ where } P(\bar{u}) ::= S \in D. \]

This yields for a recursive program \( S \) the semantics \( M[S] \) and \( M_{sp}[S] \).

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,

\[ \langle P(u + 1), \sigma \rangle \rightarrow^* \langle S; u := \sigma(u), \sigma[u := u + 1] \rangle. \]
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 \( \text{Var} \) of a normal array variable \( u \) of type

\[
\text{object} \rightarrow T.
\]

Similarly, we assume for every declaration of an instance variable \( a \) of a higher type \( T_1 \times \ldots \times T_n \rightarrow T \) a declaration in \( \text{Var} \) of a normal array variable \( a \) of type

\[
\text{object} \times T_1 \times \ldots \times T_n \rightarrow T.
\]

A normal array variable of type

\[
\text{object} \rightarrow T
\]

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

\[
\text{object} \times T_1 \times \ldots \times T_n \rightarrow T
\]

in the recursive program will represent an instance variable of the corresponding object-oriented program of type \( T_1 \times \ldots \times T_n \rightarrow T \).

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

\[
\mathcal{D}_{\text{object}} \rightarrow (\text{IVar} \rightarrow \mathcal{D}_T),
\]

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

\[
\text{Var} \rightarrow (\mathcal{D}_{\text{object}} \rightarrow \mathcal{D}_T),
\]

and for instance array variables of type \( T_1 \times \ldots \times T_n \rightarrow T \) the state \( \sigma \) is of type

\[
\mathcal{D}_{\text{object}} \rightarrow ((\text{Var} \rightarrow (\mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \rightarrow \mathcal{D}_T))),
\]

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

\[
\text{Var} \rightarrow (\mathcal{D}_{\text{object}} \times \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \rightarrow \mathcal{D}_T).
\]

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

- \( \Theta(\sigma)(x) = \sigma(x) \), for every normal variable \( x \),
- \( \Theta(\sigma)(z)(o) = \sigma(o)(z) \), for every object \( o \in \mathcal{D}_{\text{object}} \) and normal array variable \( z \) of type \( \text{object} \rightarrow T \) on the right-hand side of the equation corresponding to an instance variable \( z \) of a basic type \( T \) on the left-hand side of the equation,
Verification of Object-Oriented Programs: a Transformational Approach

\[ \Theta(\sigma)(a)(o, d_1, \ldots, d_n) = \sigma(o)(d_1, \ldots, d_n), \]
for every object \( o \in \mathcal{D}_{\text{object}} \) and normal array variable \( a \) of type \( \text{object} \times \mathcal{T}_1 \times \ldots \times \mathcal{T}_n \to \mathcal{T} \) on the right-hand side of the equation corresponding to an instance array variable \( a \) of type \( \mathcal{T}_1 \times \ldots \times \mathcal{T}_n \to \mathcal{T} \) on the left-hand side of the equation., and \( d_i \in \mathcal{D}_{\mathcal{T}_i} \), 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 \( x \),
- \( \Theta(x) \equiv x[\text{this}] \), for every instance variable \( x \) of a basic type,
- \( \Theta(a[s_1, \ldots, s_n]) \equiv a[\text{this}, \Theta(s_1), \ldots, \Theta(s_n)] \), for every instance array variable \( a \).

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

**Lemma 2. (Translation)** For all proper states \( \sigma \) the following holds.

(i) For all expressions \( s \),

\[ \sigma(s) = \Theta(\sigma)(\Theta(s)), \]

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 \( u \),

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

**Proof.** By straightforward induction on the structure of \( s \) and case analysis on the structure of \( u \). \qed

Next, we extend by structural induction the transformation \( \Theta \) 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.

- \( \Theta(u := s) \equiv \Theta(u) := \Theta(s) \),
- \( \Theta(s.m(s_1, \ldots, s_n)) \equiv \text{if } \Theta(s) \neq \text{null } \to m(\Theta(s), \Theta(s_1), \ldots, \Theta(s_n)) \text{ fi} \),
- \( \Theta(S_1; S_2) \equiv \Theta(S_1); \Theta(S_2) \),
- \( \Theta(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \equiv \text{if } \Theta(B) \text{ then } \Theta(S_1) \text{ else } \Theta(S_2) \text{ fi} \),
- \( \Theta(\text{while } B \text{ do } S \text{ od}) \equiv \text{while } \Theta(B) \text{ do } \Theta(S) \text{ od} \),
- \( \Theta(\text{if } B \to \text{ S fi}) \equiv \text{if } \Theta(B) \to \Theta(S) \text{ fi} \),
- \( \Theta(\text{begin local } \bar{u} := i; S \text{ end}) \equiv \text{begin local } \bar{u} := \Theta(i); \Theta(S) \text{ end} \),

where \( \Theta(i) \) denotes the result of applying \( \Theta \) to the sequence of expressions \( i \).
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
\[
m(\texttt{this}, u_1, \ldots, u_n) :: \Theta(S).
\]
So the distinguished normal variable \( \texttt{this} \) is added as an additional formal parameter of the procedure \( m \). This way the set \( D \) of method declarations is transformed into the set
\[
\Theta(D) = \{m(\texttt{this}, u_1, \ldots, u_n) :: \Theta(S) \mid m(u_1, \ldots, u_n) :: S \in D\}
\]
of the corresponding procedure declarations.

**Example 4.** Consider the object-oriented program
\[
S \equiv y.add(1); \ y.add(2),
\]
where \( y \) is a normal variable of type \texttt{object}, in the context of the declaration
\[
D = \{add(x) :: sum := sum + x\},
\]
where the formal parameter \( x \) is of type \texttt{integer} and \( sum \) is an instance variable, also of type \texttt{integer}. Then the transformation \( \Theta \) yields
\[
\Theta(S) \equiv \texttt{if} \ y \neq \texttt{null} \rightarrow \text{add}(y, 1) \texttt{fi}; \ \texttt{if} \ y \neq \texttt{null} \rightarrow \text{add}(y, 2) \texttt{fi}
\]
and
\[
\Theta(D) = \{add(\texttt{this}, x) :: sum[\texttt{this}] := sum[\texttt{this}] + x\}
\]
as the corresponding recursive program. \( \square \)

### 5.3 Correctness proof

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

**Lemma 3.** (Transformation) For all object-oriented programs \( S \), all sets of method declarations \( D \), all proper states \( \sigma \), and all proper or fail states \( \tau \),
\[
< S, \sigma > \rightarrow^* < E, \tau > \iff < \Theta(S), \Theta(\sigma) > \rightarrow^* < E, \Theta(\tau) > .
\]

**Proof.** We prove only the \( (\Rightarrow) \) direction. We proceed by induction on the number of the axiom and rule applications used in the computation \( < S, \sigma > \rightarrow^* < E, \tau > . \)
The only non-trivial case arises when \( S \) begins with a method call, that is, is of the form \( s.m(\overline{t}); S_1 \). By the assumption,

\[
< s.m(\overline{t}); S_1, \sigma > \rightarrow < \begin{local} \text{this}, \overline{u} := s, \overline{t}; \ S \end{local}; S_1, \sigma > \rightarrow^* < E, \tau >,
\]

where \( \sigma(s) \neq \text{null} \) and \( m(\overline{u}) :: S \in D \). So by the induction hypothesis and definition of \( \Theta \),

\[
< \Theta(\begin{local} \text{this}, \overline{u} := s, \overline{t}; \ S \end{local}); \Theta(S_1), \Theta(\sigma) > \rightarrow^* < E, \Theta(\tau) >.
\]

Note that

\[
\Theta(s.m(\overline{t}); S_1) \equiv \begin{if} \Theta(s) \neq \text{null} \rightarrow m(\Theta(s), \Theta(\overline{t})) \end{if} \Theta(S_1),
\]

By the Translation Lemma 2(i), we have \( \Theta(\sigma)(\Theta(s)) \neq \text{null} \). So by definition of the semantics of recursive programs and definition of \( \Theta \),

\[
< \begin{if} \Theta(s) \neq \text{null} \rightarrow m(\Theta(s), \Theta(\overline{t})) \end{if} \Theta(S_1), \Theta(\sigma) > \rightarrow^*< \Theta(\begin{local} \text{this}, \overline{u} := s, \overline{t}; \ S \end{local}); \Theta(S_1), \Theta(\sigma) >,
\]

which concludes the proof. \( \square \)

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

**Theorem 1. (Correctness of \( \Theta \))** For all object-oriented programs \( S \), all sets of method declarations \( D \), and all proper states \( \sigma \) the following holds:

\( \begin{align*}
(i) \ & \Theta(\mathcal{M}[S](\sigma)) = \mathcal{M}[\Theta(S)][\Theta(\sigma)], \\
(ii) \ & \Theta(\mathcal{M}_p[S](\sigma)) = \mathcal{M}_p[\Theta(S)][\Theta(\sigma)],
\end{align*} \)

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 3 and the fact that each ‘normal’ state is of the form \( \Theta(\tau) \) for some ‘object-oriented’ state \( \tau \). \( \square \)

### 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 $\text{object}$, $s_1, \ldots, s_n$ are global expressions of type $T_1, \ldots, T_n$, and $a$ is an array instance variable of type $T_1 \times \ldots \times T_1 \rightarrow T$ 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 5.** Consider a normal integer variable $i$, a normal variable $x$ of type $\text{object}$, a normal array variable of type $\text{integer} \rightarrow \text{object}$, and an instance variable $\text{next}$ of type $\text{object}$. Using them we can generate the following global expressions:

- $\text{next}$, $\text{next.next}$, etc.,
- $x.next$, $x.next.next$, etc.,
- $a[i].\text{next}$,

all of type $\text{object}$. In contrast, $\text{next}.x$ 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 $\text{next.next}$ refers to the object that can be reached by ‘moving’ to the object denoted by the value of $\text{next}$ of the current object $\text{this}$ and evaluating the value of its variable $\text{next}$.

We define the semantics of global expressions by extending the semantics of expressions given in Subsection 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 $T$,

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

where $\sigma(s) = o$.

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

Note that this semantics also provides meaning to global expressions of the form $\text{null}.u$. However, such expressions are meaningless when specifying correctness of programs because the local state of the $\text{null}$ object can never be reached in computations starting in a proper state $\sigma$ such that $\sigma(\text{this}) \neq \text{null}$ (see the Safety Lemma 1).

**Example 6.** If $x$ is an object variable and $\sigma$ a proper state such that $\sigma(x) \neq \text{null}$, then for all simple instance variables $y$

$$\sigma(x.y) = \sigma(\sigma(x))(y).$$

$\square$
Assertions are constructed from global Boolean expressions by adding quantification over normal variables. We use $p, q$ as typical letters for assertions. For a state $\sigma$ and a assertion $p$ we write $\sigma \models p$ if $\sigma$ 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 $s[x := t]$ 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, substitution yields

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

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 $\sigma$ 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 $\sigma$. 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 $a \neq b$, then
  $$s[u := t] \equiv a[s_1[u := t], \ldots, s_n[u := t]],$$
- if $s \equiv a[s_1, \ldots, s_n]$ for some array $a$ and $u \equiv a[t_1, \ldots, t_n]$ then
  $$s[u := t] \equiv \text{if } \bigwedge_{i=1}^n s_i' = t_i \text{ then } t \text{ else } a[s_1', \ldots, s_n'] \quad \text{fi}$$
  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

$$\text{if } \bigwedge_{i=1}^n s_i' = t_i \text{ then } \ldots \text{ else } \ldots \quad \text{fi}$$

checks whether, for any given proper state $\sigma$, 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 $\sigma$ 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  
\[ x[u := t] \equiv x, \]

– if $s \equiv e.u$ and $u$ is a simple instance variable then 
\[ s[u := t] \equiv \text{if } e' = \text{this then } t \text{ else } e'.u \text{ fi}, \]
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
\[ s[u := t] \equiv \text{if } e' = \text{this} \land \bigwedge_{i=1}^n s'_i = t_i \text{ then } t \text{ else } e'.a(s'_1, \ldots, s'_n) \text{ fi}, \]
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 7.** Suppose that $s \equiv \text{this}.u$. Then
\[
\text{this}.u[u := t] \\
\equiv \text{if this}[u := t] = \text{this then } t \text{ else } \text{fi} \\
\equiv \text{if this = this then } t \text{ else } \text{fi}.
\]

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

Next, suppose that $s \equiv \text{this}.a[x]$, where $x$ is a simple variable. Then
\[
\text{this}.a[x][a[x] := t] \\
\equiv \text{if this}[a[x] := t] = \text{this} \land x[a[x] := t] = x \text{ then } t \text{ else } \text{fi} \\
\equiv \text{if this = this} \land x = x \text{ then } t \text{ else } \text{fi}.
\]

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

Finally, for a simple instance variable $u$ and a normal object variable $x$ we have
\[
x.u[u := t] \\
\equiv \text{if } x[u := t] = \text{this then } t \text{ else } x[u := t].u \text{ fi} \\
\equiv \text{if } x = \text{this then } t \text{ else } x.u \text{ fi}.
\]
The substitution operation is then extended to assertions by properly taking care of quantification, as usual in logic. We have the following lemma that relates for instance variables the effect of substitution to the state update.

**Lemma 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 $\sigma$ the following holds:

(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 $p$. \qed

### 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 $M$ and $M_{sp}$, we formalize these two interpretations of correctness formulas uniformly as set theoretic inclusions (cf. [3]):

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

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

**Example 8.** Consider again the program $S \equiv \text{this}.\text{find}(z)$ of Example 2 for finding an object in a linked list. To specify the desired effect of the there declared method $\text{find}$ we introduce a normal array variable $a$ of type $\text{integer} \rightarrow \text{object}$ that stores a linked list of objects, as expressed by the assertion

$$p_0 \equiv \forall i \geq 0 : a[i].\text{next} = a[i + 1].$$

Then we take as precondition

$$p \equiv \text{this} = a[0] \land p_0$$

and as postcondition

$$q \equiv \exists i \geq 0 : z = a[i].$$

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.
However, it does not hold in the sense of strong partial correctness because the list might contain \texttt{null} objects. To avoid this we strengthen the precondition by adding the assertion
\[ p_1 \equiv \forall i \geq 0 : a[i] \neq \texttt{null}. \]
Then \( \{ p \land p_1 \} \ S \ \{ q \} \) holds in the sense of strong partial correctness. \hfill \square

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**
\[
\{ p[u := t]\} \ \ u := t \ \{ p \}
\]
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**
\[
\{ p \land s \neq \texttt{null}\} \ s.m(\texttt{t}) \ \{ q \} \\
\{ p \} \ s.m(\texttt{t}) \ \{ q \}
\]

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:

\[
\begin{align*}
\{ p \} \ \texttt{begin local this, } \bar{u} := s, \bar{t}; S \ \texttt{end} \ \{ q \} \\
\{ p \} \ s.m(\bar{t}) \ \{ q \}
\end{align*}
\]
where \( m(\bar{u}) :: S \in D \).

**Example 9.** We prove the partial correctness formula
\[
\{ \texttt{true} \} \ \texttt{null}.m \ \{ \texttt{false} \},
\]
where \( m :: \texttt{skip} \in D \). First, we have
\[
\{ \texttt{false} \} \ \texttt{begin local this := null; skip end} \ \{ \texttt{false} \},
\]
so by the above copy rule we get
\[
\{ \texttt{false} \} \ \texttt{null}.m \ \{ \texttt{false} \}.
\]
The desired conclusion now follows by the above weakening rule and the consequence rule. \hfill \square
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

\[
\{ p \} \ s.m(\bar{t}) \ \{ q \} \vdash_{PO} \{ p \} \ \begin{array}{l}
\text{begin local this, } \bar{u} := s, \bar{t}; \ S \end{array} \ \{ q \}
\]

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**

\[
\{ p_1 \} \ s_1.m_1(\bar{t}_1) \ \{ q_1 \}, \ldots, \{ p_n \} \ s_n.m_n(\bar{t}_n) \ \{ q_n \} \vdash_{PO} \{ p \} \ S \ \{ q \},
\]

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 the proof system \( PO \) 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**

\[
\begin{align*}
\{p_1\} & \ s_1.m_1(\bar{t}_1) \ {\{q_1\}} \ \ldots \ \ldots \ {\{p_n\}} \ s_n.m_n(\bar{t}_n) \ {\{q_n\}} +_{\text{SPO}} \ {\{p\}} \ S \ {\{q\}}, \\
\{p_1\} & \ s_1.m_1(\bar{t}_1) \ {\{q_1\}} \ \ldots \ \ldots \ {\{p_n\}} \ s_n.m_n(\bar{t}_n) \ {\{q_n\}} +_{\text{SPO}} \\
{\{p_i\}} & \ \text{begin local this,} \ \bar{u}_i := s_i, \bar{t}_i; \ S_i \ \text{end} \ {\{q_i\}}, \ i \in \{1, \ldots, n\} \\
(*) & \ p_i \rightarrow s_i \neq \text{null}, \ i \in \{1, \ldots, n\} \\
\{p\} & \ S \ {\{q\}}
\end{align*}
\]

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 10.** Let \( m :: \text{skip} \in D \). Without the premises \((*)\), we could derive from

\[
\{\text{true}\} \ \text{null}.m \ \{\text{true}\} \vdash \{\text{true}\} \ \text{begin local this} := \text{null}; \ \text{skip} \ \text{end} \ \{\text{true}\}.
\]

the correctness formula

\[
\{\text{true}\} \ \text{null}.m \ \{\text{true}\}.
\]

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 the proof system \( SPO \) 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 1, 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 5. (Semantic invariance)** Let \( N \) 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

\[
N[S][\sigma[\bar{z} := \bar{d}]](\tau := \bar{d}) = \{\tau[\bar{z} := \bar{d}] \mid \tau \in N[S][\sigma]\}.
\]

**Proof.** The proof proceeds by a straightforward induction on the length of the computation. \( \square \)
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**

\[
\begin{align*}
\{p_1\} P_1(\bar{x}_1) \{q_1\}, \ldots, \{p_n\} P_n(\bar{x}_n) \{q_n\} & \vdash \{p\} S \{q\}, \\
\{p_1\} P_1(\bar{x}_1) \{q_1\}, \ldots, \{p_n\} P_n(\bar{x}_n) \{q_n\} & \vdash \\
\{p_i\} \text{begin local } \bar{u}_i := \bar{x}_i; S_i \text{ end } \{q_i\}, \ i \in \{1, \ldots, n\}
\end{align*}
\]

where \(P_i(\bar{u}_i) :: S_i \in D\) and \(\text{var}(\bar{x}_i) \cap \text{var}(D) = \emptyset\) for \(i \in \{1, \ldots, n\}\).

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 the proof system \(PR\) by the recursion III rule 15.

**PROOF SYSTEM \(SPR^+\)** for strong partial correctness of recursive programs:

This system is obtained by extending the proof system \(SPR\) 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 Subsection 5.2 to global expressions introduced in Subsection 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 6. (Assertion)** For all assertions \(p\) and all proper states \(\sigma\)

\[\sigma \models p \iff \Theta(\sigma) \models \Theta(p).\]
Proof. The straightforward proof proceeds by induction on the structure of \( p \).

Corollary 1. (Translation I) For all correctness formulas \( \{p\} S \{q\} \), where \( S \) is an object-oriented program,

\[
\vdash \{p\} S \{q\} \text{ iff } \vdash \{\Theta(p)\} \Theta(S) \{\Theta(q)\},
\]

and

\[
\vdash_{sp} \{p\} S \{q\} \text{ iff } \vdash_{sp} \{\Theta(p)\} \Theta(S) \{\Theta(q)\}.
\]

Proof. It follows directly by the Assertion Lemma 6 and the Correctness Theorem 1.

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

\[
\Theta(A) = \{\Theta(p)\} m(\Theta(s), \Theta(\ell)) \{\Theta(q)\} | \{p\} s.m(\ell) \{q\} \in A.
\]

Lemma 7. (Translation of Correctness Proofs of Translated Method Calls)

Let \( A \) be a given set of assumptions about method calls. Then

\[
A \vdash_{AR} \{p\} s.m(\ell) \{q\} \text{ iff } \Theta(A) \vdash_{AR} \{\Theta(p)\} m(\Theta(s), \Theta(\ell)) \{\Theta(q)\},
\]

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 a straightforward induction on the length of the derivation.

In order to prove the 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. (Normal Form Partial Correctness Failure Statements)

Let \( A \) be a given set of assumptions about procedure calls. If

\[
A \vdash_{PR} \{p\} \text{ if } B \rightarrow S \text{ fi } \{q\}
\]

then

\[
A \vdash_{PR} \{p \land B\} S \{q\}.
\]
**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

\[ A \vdash_{PR} \{ p' \} \text{ if } B \rightarrow S \text{ fi } \{ q \} \]

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

\[ A \vdash_{PR} \{ \exists x : (p' \land B) \} S \{ q \} \]

Since \( x \) does not occur in \( B \), the precondition is logically equivalent to \( (\exists x : p') \land B \), 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 9. (Translation of Partial Correctness Proofs of Method Calls)**

Let \( A \) be a given set of assumptions about method calls. Then

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

iff

\[ \Theta(A) \vdash_{PR} \{ \Theta(p) \} \text{ if } \Theta(s) \neq \text{null } \rightarrow m(\Theta(s), \Theta(\bar{t})) \text{ fi } \{ \Theta(q) \} \].

**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 \( \Theta(A) \),

\[ \{ \Theta(p) \} m(\Theta(s), \Theta(\bar{t})) \{ \Theta(q) \} \in \Theta(A) \],

so

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

By a trivial application of the consequence rule, we get

\[ \Theta(A) \vdash_{PR} \{ \Theta(p) \land \Theta(s) \neq \text{null} \} m(\Theta(s), \Theta(\bar{t})) \{ \Theta(q) \} \].

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

\[ A \vdash_{PO} \{ p \land s \neq \text{null} \} s.m(\bar{t}) \{ q \} \].

By the induction hypothesis,

\[ \Theta(A) \vdash_{PR} \{ \Theta(p) \land \Theta(s) \neq \text{null} \} \text{ if } \Theta(s) \neq \text{null } \rightarrow m(\Theta(s), \Theta(\bar{t})) \text{ fi } \{ \Theta(q) \} \].
By Lemma 8, it follows that
\[ \Theta(A) \vdash_{PR} \{ \Theta(p) \land \Theta(s) \neq \text{null} \land \Theta(s) \neq \text{null} \} \ m(\Theta(s), \Theta(\bar{t})) \ \{ \Theta(q) \} , \]
so by the consequence and failure rules we get the desired result, the right-hand side of the statement of the lemma.

(\Rightarrow) 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
\[ \Theta(A) \vdash_{PR} \{ \Theta(p) \land \Theta(s) \neq \text{null} \} \ m(\Theta(s), \Theta(\bar{t})) \ \{ \Theta(q) \} . \]
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 7, we get
\[ A \vdash_{AR} \{ p \land s \neq \text{null} \} \ s.m(\bar{t}) \ \{ q \} . \]
Applying the weakening rule we get the desired result, the left-hand side of the statement of the lemma. \[ \square \]

**Lemma 10. (Translation of Strong Partial Correctness Proofs of Method Calls)**

Let \( A \) be a given set of assumptions about method calls such that \( p' \rightarrow s' \neq \text{null} \) holds for all \( \{ p' \} s'.m'(\bar{t}') \ \{ q' \} \in A \). Then
\[ A \vdash_{SPo} \{ p \} \ s.m(\bar{t}) \ \{ q \} \]
iff
\[ \Theta(A) \vdash_{SPR} \{ \Theta(p) \} \ \text{if} \ \Theta(s) \neq \text{null} \rightarrow m(\Theta(s), \Theta(\bar{t})) \ \text{fi} \ \{ \Theta(q) \} . \]

**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 \( p \rightarrow s \neq \text{null} \) holds. By definition of \( \Theta(A) \),
\[ \{ \Theta(p) \} m(\Theta(s), \Theta(\bar{t})) \ \{ \Theta(q) \} \in \Theta(A) . \]
Furthermore, by the Assertion Lemma 6, we have \( \Theta(p) \rightarrow \Theta(s) \neq \text{null} \). 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 \( \Theta(p) \rightarrow \Theta(s) \neq \text{null} \) holds and the rule is applied to
\[ \Theta(A) \vdash_{SPR} \{ \Theta(p) \} \ m(\Theta(s), \Theta(\bar{t})) \ \{ \Theta(q) \} . \]
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 7,
\[ A \vdash_{AR} \{ p \} \ s.m(\bar{t}) \ \{ q \} . \]
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 11. (Homomorphism)** For all global expressions or assertions $p$, all expressions $t$ of the programming language, and all simple or subscripted variables $u$,

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

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

$$
\Theta(s[u := t]) \equiv \Theta(s)[u[this] := \Theta(t)]
$$

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

$$
\begin{align*}
\Theta(e.u[u := t]) & \equiv \{ \text{by definition of the substitution } [u := t] \} \\
\Theta(\text{if } e[u := t] = \text{this then } t \text{ else } e[u := t], u \text{ fi}) & \equiv \{ \text{by definition of } \Theta \} \\
\text{if } \Theta(e[u := t]) = \text{this} \ & \ \Rightarrow \ \text{else } \Theta(e[u := t], u) \text{ fi} \\
& \equiv \{ \text{by definition of } \Theta \} \\
\text{if } \Theta(e[u := t]) = \text{this} \ & \ \Rightarrow \ \text{else } u[\Theta(e[u := t])] \text{ fi} \\
& \equiv \{ \text{by induction hypothesis about } e \} \\
\text{if } \Theta(e)[u[this] := \Theta(t)] = \text{this} \ & \ \Rightarrow \ \text{else } u[\Theta(e)[u[this] := \Theta(t)]] \text{ fi} \\
& \equiv \{ \text{by definition of the substitution } [u[this] := \Theta(t)] \} \\
u[\Theta(e)][u[this] := \Theta(t)] & \equiv \{ \text{by definition of } \Theta \} \\
\Theta(e.u)[u[this] := \Theta(t)] & \equiv \{ \text{by definition of the substitution } [u := t] \} \\
\end{align*}
$$

**Lemma 12. (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

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

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 ⊢ on the left-hand-side denotes provability in the proof system SPO, and ⊢ on the right-hand-side denotes provability in the proof system SPR. Additionally, we assume that \( p' \rightarrow s \neq \text{null} \) holds for all \( \{p'\} s.m(i) \{q'\} \in A \).

**Proof.** The proof proceeds by induction on the length of the derivation. The case of an assignment statement follows by the Homomorphism Lemma 11. The case of a method call follows by the Translation Lemma 10. 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 6 is used. \( \square \)

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

**Theorem 2. (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 11. The case of the recursion rules is taken care of by the Translation Lemma 12. The case of the other axioms and rules follows immediately from the induction hypothesis (using the Assertion Lemma 6 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. \( \square \)

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 $\sigma \vdash \sigma'$, for $V \subseteq \text{Var}$, we denote 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 $\sigma[[x := \sigma'(y)]]$ 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 re-naming every variable $x_i$ by $y_i$, for $i \in \{1, \ldots, k\}$. We have the following substitution lemma corresponding to the Substitution Lemma 4.

Lemma 13. (Substitution) We have

$$
\sigma \vdash p[[x := \bar{y}]] \text{ iff } \sigma[[x := \sigma'(y)]] \vdash p.
$$

Proof. The proof proceeds by a straightforward induction on the structure of $p$. \hfill \Box

The weakest precondition $\text{WP}(S, p)$ for partial correctness denotes the set

$$
\{ \sigma \mid \mathcal{M}[S](\sigma) \subseteq [p] \}.
$$

Similarly, the weakest precondition $\text{WP}_p(S, p)$ for strong partial correctness denotes the set

$$
\{ \sigma \mid \mathcal{M}_p[S](\sigma) \subseteq [p] \}.
$$

The above predicates satisfy the following equations.

Lemma 14. (Weakest precondition calculus) Let $W$ stand for $\text{WP}$ or $\text{WP}_p$. The weakest preconditions of the statements of the kernel language satisfy the following (standard) equations.

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

Failure statements satisfy

$$
\text{WP}(\text{if } B \rightarrow S, p) = ([B] \cap \text{WP}(S, p)) \cup [\neg B]
$$

and

$$
\text{WP}_p(\text{if } B \rightarrow S, p) = [B] \cap \text{WP}(S, p).
$$

Finally, block statements satisfy

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

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

\[
W(\text{begin local } \bar{u} := \bar{t}; S \text{ end}, p) \\
= W(\bar{u} := \bar{t}; S; \bar{u} := \sigma(\bar{u}), p) \\
= W(W(\bar{u} := \sigma(\bar{u})), \bar{u} := \bar{t}; S, p) \\
= W(\bar{u} := \bar{t}; S, p).
\]

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

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

Note that by definition,

\[
WP(S, \bar{x} = \bar{y}) = \{\sigma \mid M[S](\sigma) \subseteq \{\sigma[\bar{x} := \sigma(\bar{y})]\}\}
\]

and

\[
WP_{sp}(S, \bar{x} = \bar{y}) = \{\sigma \mid M_{sp}[S](\sigma) \subseteq \{\sigma[\bar{x} := \sigma(\bar{y})]\}\}.
\]

These predicates describe the graphs of the deterministic functions \(M[S]\) and \(M_{sp}[S]\) 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 \(\text{nat}\) 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 \(\text{nat} \rightarrow T\) 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 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 \(\text{nat}\).

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 \(\text{code}(n, z)\) defined by \(h[n] = z\), where \(z\) is a (simple) variable, directly expresses that the variable \(n\) (of type \(\text{nat}\)) stores an integer representation of the value of \(z\). In order to express a similar assertion \(\text{code}(n, a)\), where \(a\) is an array variable, we assume in the assertion language the following arithmetic operations:

- \(< \bar{n} >\) denotes the natural number encoding the sequence of natural numbers \(\bar{n}\),
- \(n(i)\) denotes the \(i\)th 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

\[
|n| = l \land a[h[n(1)], \ldots, h[n(l)]] = h[n(l+1)]
\]

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

\[
\forall 1 \leq k \leq |n|: app(n(k), a).
\]

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

\[
|n| = k \land \bigwedge_{i=1}^{k} code(n(i), z_i).
\]

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 \( \sigma \) 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 predicate \( comp \) which denotes the set

\[
\{ (n, m) | \forall \sigma, \sigma' : code(n, \bar{x}) \land \sigma' \models code(m, \bar{x}) \text{ implies } \sigma[\bar{x} := \sigma' (\bar{x})] \in M[S](\sigma) \}.
\]

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

**Lemma 15. (Closure of \( comp \))** For all states \( \sigma \) and \( \sigma' \) we have that \( \sigma' \in M[S](\sigma) \) implies \( comp(n, m) \), for some pair of numbers \( n \) and \( m \) such that \( \sigma \models code(n, \bar{x}) \) and \( \sigma' \models code(m, \bar{x}) \).

We proceed with the introduction of the (unary) arithmetic predicate \( fail \) which denotes the set

\[
\{ n | \forall \sigma : \sigma \models code(n, \bar{x}) \text{ implies } fail \in M[S](\sigma) \}.
\]

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

**Lemma 16. (Closure of \( fail \))** For every state \( \sigma \) such that \( fail \in M[S](\sigma) \) we have \( \sigma \models code(n, \bar{x}) \) and \( fail(n) \), 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 \( comp \) and \( fail \) 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 17. (Expressibility of the most general weakest preconditions) For the assertion
\[ p \equiv \forall n : \forall m : (\text{code}(n, \bar{x}) \land \text{comp}(n, m)) \rightarrow \text{code}(m, \bar{y}) \]
we have
\[ \text{WP}(S, \bar{x} = \bar{y}) = [p] \]
and
\[ \text{WP}_{\text{sp}}(S, \bar{x} = \bar{y}) = [p \land \forall n : \text{code}(n, \bar{x}) \rightarrow \neg \text{fail}(n)] \].

Proof. We prove first the first equation. Let \( \sigma \in \text{WP}(S, \bar{x} = \bar{y}) \), i.e., \( \mathcal{M}[S](\sigma) \subseteq \{ \sigma[\bar{x} := \sigma(\bar{y})] \} \). In order to prove \( \sigma \models p \), let \( \sigma \models \text{code}(n, \bar{x}) \land \text{comp}(n, m) \), for some arbitrary (constants) \( n \) and \( m \). Further, let \( \sigma' \models \text{code}(m, \bar{x}) \), for some \( \sigma' \) (note that the encoding of the variables \( \bar{x} \) is assumed to be surjective). By definition of \( \text{comp} \), we have \( \sigma'[\bar{x} := \sigma'(\bar{x})] \in \mathcal{M}[S](\sigma) \). Because \( S \) is deterministic it follows that \( \sigma[\bar{x} := \sigma'(\bar{x})] = \sigma[\bar{x} := \sigma(\bar{y})] \). Since the evaluation of \( \text{code}(m, \bar{x}) \) only depends on the interpretation of the variables \( \bar{x} \), \( \sigma' \models \text{code}(m, \bar{x}) \) implies \( \sigma[\bar{x} := \sigma'(\bar{x})] \models \text{code}(m, \bar{x}) \). That is, \( \sigma[\bar{x} := \sigma(\bar{y})] \models \text{code}(m, \bar{x}) \). We conclude by Lemma 13 that \( \sigma \models \text{code}(m, \bar{y}) \).

Next let \( \sigma \models p \). In order to prove \( \sigma \in \text{WP}(S, \bar{x} = \bar{y}) \), i.e., \( \mathcal{M}[S](\sigma) \subseteq \{ \sigma[\bar{x} := \sigma(\bar{y})] \} \), let \( \sigma' \in \mathcal{M}[S][\sigma] \). By Lemma 15, it follows that \( \text{comp}(n, m) \), for some \( n \) and \( m \) such that \( \sigma \models \text{code}(n, \bar{x}) \) and \( \sigma' \models \text{code}(m, \bar{x}) \). From \( \sigma \models p \) it follows that \( \sigma \models \text{code}(m, \bar{y}) \). From Lemma 13 it then follows that \( \sigma[\bar{x} := \sigma(\bar{y})] \models \text{code}(m, \bar{x}) \). So by the definition of \( \text{comp} \) we have \( \sigma[\bar{x} := \sigma(\bar{y})] \in \mathcal{M}_{\text{sp}}[S](\sigma) \). Since \( S \) is deterministic we conclude that \( \sigma' = \sigma[\bar{x} := \sigma(\bar{y})] \).

For the second equation, it suffices to observe that by Lemma 16, \( \text{fail} \in \mathcal{M}_{\text{sp}}[S][\sigma] \) implies \( \sigma \models \exists n : \text{code}(n, \bar{x}) \land \text{fail}(n) \). On the other hand, by definition of \( \text{fail} \) it immediately follows that \( \sigma \models \exists n : \text{code}(n, \bar{x}) \land \text{fail}(n) \) implies \( \text{fail} \in \mathcal{M}_{\text{sp}}[S][\sigma] \). We conclude that \( \text{fail} \notin \mathcal{M}_{\text{sp}}[S][\sigma] \) iff \( \sigma \models \forall n : \text{code}(n, \bar{x}) \rightarrow \neg \text{fail}(n) \).

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 18. (Expressibility of divergence/failure) For the assertion
\[ p \equiv \forall n : \forall m : \text{code}(n, \bar{x}) \rightarrow \neg \text{comp}(n, m) \]
we have
\[ \text{WP}(S, \text{false}) = [p] \]
and
\[ \text{WP}_{\text{sp}}(S, \text{false}) = [p \land \forall n : \text{code}(n, \bar{x}) \rightarrow \neg \text{fail}(n)] \].

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 \( \sigma \in \text{WP}(S, \text{false}) \), i.e., \( \mathcal{M}[S](\sigma) = \emptyset \). In order to prove \( \sigma \models p \), let \( \sigma \models \text{code}(n, \bar{x}) \land
Proof. Let \( \models S\{p\} \) 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

\[
\{q[\bar{x} := \bar{y}] \land \text{WP}_{sp}(S, \bar{x} = \bar{y})\} S \{q[\bar{x} := \bar{y}] \land \bar{x} = \bar{y}\}.
\]

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

\[
\{q[\bar{x} := \bar{y}] \land \text{WP}_{sp}(S, \bar{x} = \bar{y})\} S \{q\}.
\]

Next we apply the auxiliary rule A3:

\[
\{\exists \bar{y} : q[\bar{x} := \bar{y}] \land \text{WP}_{sp}(S, \bar{x} = \bar{y})\} S \{q\}.
\]

By definition of the weakest precondition and \( \models S\{p\} \), it follows that \( p \) implies the above precondition (an application of the consequence rule thus gives us the desired correctness formula): Let \( \sigma = p \). It follows from \( \models S\{p\} \) that \( M_{sp}[S](\sigma) \subseteq [q] \), i.e., \( M_{sp}[S](\sigma) = 0 \) or \( \sigma' \in M_{sp}[S](\sigma) \), for some proper state \( \sigma' \). First we consider the case that \( M_{sp}[S](\sigma) = 0 \). From Lemma 5 it follows that \( M_{sp}[S](\sigma') = \)}
∅, for every σ′ such \( \sigma' = \text{var} \setminus \bar{y} \) σ. Further, since q is consistent, we have \( \sigma' \models q[\bar{x} := \bar{y}] \), for some σ′ such \( \sigma' = \text{var} \setminus \bar{y} \) σ. Summarizing, we have

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

for some σ′ such that \( \sigma' = \text{var} \setminus \bar{y} \) σ. From which we conclude that

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

Next we consider the case that \( \sigma' \in \mathcal{M}_{sp}[S](\sigma) \), for some proper state \( \sigma' \). From Lemma 5 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 13 it then follows that \( \sigma'[\bar{y} := \sigma'(\bar{x})] \models q[\bar{x} := \bar{y}] \). Since \( \sigma = \text{var} \setminus \bar{y} \) \( \sigma' \) 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

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

that is,

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

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

\[
\{ WP_{sp}(P_i(\bar{t}_i), \text{false}) \} P_i(\bar{t}_i) \{ \text{false} \} \text{ and } \{ WP_{sp}(P_i(\bar{t}_i), \bar{x} = \bar{y}) \} P_i(\bar{t}_i) \{ \bar{x} = \bar{y} \},
\]

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 18. Let \( A \) denote the set of these correctness formulas.

**Lemma 20. (Completeness assumptions)** A) We have

\[
\models_{sp}(p) S \{q\} \text{ implies } A \vdash_{SPR} \{p\} S \{q\}.
\]

**Proof.** The proof proceeds by induction on the structure of the statement \( S \). Distinguishing between \( \models_{sp}(p) S \{\text{false}\} \) and \( \models_{sp}(p) S \{q\} \), where \( q \) is consistent, by definition of \( WP_{sp}(S, \text{false}) \) and the above Lemma 19, it suffices to prove

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

where \( r \) denotes the assertion \text{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 14. We consider therefore the non-standard case that \( S \) denotes
a block statement `begin local ¯u := ¯t; S1 end`. We introduce a sequence ¯z of fresh variables corresponding to the local variables ¯u. By the semantics of block statements, it follows that

\[| = sp\{ ¯z = ¯u \land WP_sp(S, r)\} \} ¯u := ¯t; S1 \} r[¯u := ¯z]\].

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

\[\{ ¯z = ¯u \land WP_sp(S, r)\} S \} r[¯u := ¯z]\].

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

\[\{ ¯z = ¯u \land WP_sp(S, r)\} S \} \bar{z} = \bar{u} \land r[\bar{u} := \bar{z}]\].

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

\[\{ ¯z = ¯u \land WP_sp(S, r)\} S \} r\].

Finally, applying the substitution rule (replacing in the precondition ¯z by ¯u) followed a trivial application of the consequence rule gives us the desired result.

We conclude with the following main completeness theorem.

**Theorem 3. (Completeness for strong partial correctness of recursive programs)**

Every strong partially correct specification \{p\} S \{q\} of a recursive program S is derivable in SPR+. Formally, \(\vdash_{SPR^+} \{p\} S \{q\}\) 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 20, we have

\[A \vdash_{SPR} \{p\} S \{q\}\].

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

\[\models_{sp} \{WP_sp(Pi(\bar{t}_i)), r\} Pi(\bar{t}_i) \} r\]

implies

\[\models_{sp} \{WP_sp(Pi(\bar{t}_i), r)\} \} begin local \bar{u}_i := \bar{t}_i; S_i end \} r\],

for every \(Pi(\bar{u}_i) :: S_i \in D\). By Lemma 20 again we have

\[A \vdash_{SPR} \{ WP_sp(Pi(\bar{t}_i), r) \} \} begin local \bar{u}_i := \bar{t}_i; S_i end \} r\],

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, both to more powerful programming languages 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

- introduction of the global expressions already in Subsection 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 Subsection 4.1,
- extension of the notion of an update of a state \( \sigma[s := d] \) in Subsection 4.2 to the case of a global expression \( s \),
- extension of the definition of substitution given in Subsection 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 \( \Theta \) given in Section 5 to the considered programming language, by defining \( \Theta(s) \) for a global expression \( s \) already in Subsection 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 11, 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 \( \text{store} \) of type \( \mathbb{N} \rightarrow \text{object} \) and a variable \( \text{count} \) of type \( \mathbb{N} \), we model the object creation statement \( x := \text{new} \) by the statement

\[
\text{count} := \text{count} + 1; x := \text{store}[\text{count}].
\]

This modeling of the object activation crucially depends on \( \text{store} \) being an unbounded array variable. Further it assumes that \( \text{store} \) is injective:

\[
\forall i : \forall j : i \neq j \rightarrow \text{store}[i] \neq \text{store}[j].
\]
A drawback of this approach to object creation is that it involves an explicit reference to a particular implementation. Since an object variable 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 \( x := \text{new} \) 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 \( p[x := \text{new}] \) of an object creation statement \( x := \text{new} \) and a postcondition \( p \) which abstracts from the particular implementation of object creation.

10.3 Total correctness

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

- addition of a special state \( \bot \) 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 chapter 6 of our book [3].

11 Summary

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) is given 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 \textit{object}) and show that the transformation preserves completeness.

We finally observe that we can apply the transformational approach itself to most of the intricate and complex object-oriented features, such as exception handling and inheritance, by translating them into the core language considered in this paper and as such extend the relative completeness result to the corresponding proof rules.

Appendix

A Semantics

In the following we list the omitted transition axioms and rules that define the transition relation \( \rightarrow \).
(i) \(<\text{skip}, \sigma > \rightarrow < E, \sigma >\),
(ii) \(< u := t, \sigma > \rightarrow < E, \sigma[u := \sigma(t)] >\),
where \(u \in Var\) is a simple variable or \(u \equiv a[s_1, \ldots, s_n]\), for \(a \in Var\),
(iii) \(< \bar{x} := \bar{t}, \sigma > \rightarrow < E, \sigma[\bar{x} := \sigma(\bar{t})] >\)
\(< S_1, \sigma > \rightarrow < S_2, \tau >\)
(iv) \(< S_1; S, \sigma > \rightarrow < S_2; S, \tau >\)
(v) \(< \text{if} \; B \; \text{then} \; S_1 \; \text{else} \; S_2 \; \text{fi}, \sigma > \rightarrow < S_1, \sigma >\), where \(\sigma \models B\),
(vi) \(< \text{if} \; B \; \text{then} \; S_1 \; \text{else} \; S_2 \; \text{fi}, \sigma > \rightarrow < S_2, \sigma >\), where \(\sigma \models \neg B\),
(vii) \(< \text{while} \; B \; \text{do} \; S \; \text{od}, \sigma > \rightarrow < S, \text{while} \; B \; \text{do} \; S \; \text{od}, \sigma >\), where \(\sigma \models B\),
(viii) \(< \text{while} \; B \; \text{do} \; S \; \text{od}, \sigma > \rightarrow < E, \sigma >\) where \(\sigma \models \neg B\).

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 \(\text{free}(q)\).

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**

\[ \{ p \} \text{skip} \{ p \} \]

**AXIOM 2: ASSIGNMENT**

\[ \{ p[u := t] \} u := t \{ p \} \]
where \(u \in Var\) or \(u \equiv a[s_1, \ldots, s_n]\) and \(a \in Var\).

**AXIOM 3: PARALLEL ASSIGNMENT**

\[ \{ p[\bar{x} := \bar{t}] \} \bar{x} := \bar{t} \{ p \} \]

**RULE 4: COMPOSITION**

\[ \begin{array}{c}
\{ p \} S_1 \{ r \}, \{ r \} S_2 \{ q \} \\
\{ p \} S_1; S_2 \{ q \}
\end{array} \]

**RULE 5: CONDITIONAL**

\[ \begin{array}{c}
\{ p \land B \} S_1 \{ q \}, \{ p \land \neg B \} S_2 \{ q \} \\
\{ p \} \text{if} \; B \; \text{then} \; S_1 \; \text{else} \; S_2 \; \text{fi} \{ q \}
\end{array} \]
RULE 6: LOOP

\[
\{p \land B\} \{p\} \quad \{p\} \text{ while } B \text{ do } S \text{ od } \{p \land \neg B\}
\]

RULE 7: CONSEQUENCE

\[
p \rightarrow p_1, \{p_1\} \{q_1\}, q_1 \rightarrow q \quad \{p\} \{q\}
\]

RULE 8: FAILURE

\[
\{p \land B\} \{q\} \\
\{p\} \text{ if } B \rightarrow S \text{ fi } \{q\}
\]

RULE 9: FAILURE II

\[
p \rightarrow B, \{p\} \{q\} \\
\{p\} \text{ if } B \rightarrow S \text{ fi } \{q\}
\]

RULE 10: BLOCK

\[
\{p\} \bar{x} := \bar{t}; S \{q\} \\
\{p\} \text{ begin local } \bar{x} := \bar{t}; S \text{ end } \{q\}
\]

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

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 \(D\).

RULE A1: DISJUNCTION

\[
\{p\} \{q\}, \{r\} \{q\} \quad \{p \lor r\} \{q\}
\]

RULE A2: CONJUNCTION

\[
\{p_1\} \{q_1\}, \{p_2\} \{q_2\} \\
\{p_1 \land p_2\} \{q_1 \land q_2\}
\]

RULE A3: \(\exists\)-INTRODUCTION

\[
\{p\} \{q\} \\
\{\exists x : p\} \{q\}
\]

where \(x \notin \text{var}(D) \cup \text{var}(S) \cup \text{free}(q)\).
RULE A4: INVARIANCE

\[
\{ r \} \, S \, \{ q \} \\
\{ p \land r \} \, S \, \{ p \land q \}
\]

where \( \text{free}(p) \cap (\text{change}(D) \cup \text{change}(S)) = \emptyset \).

RULE A5: SUBSTITUTION

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

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

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

\[
\{ p[u := t] \} \, u := \{ p \}
\]

where \( u \) is a simple or subscripted instance variable.

RULE 12: WEAKENING

\[
\{ p \land s \neq \text{null} \} \, s.m(\bar{t}) \, \{ q \} \\
\{ p \} \, s.m(\bar{t}) \, \{ q \}
\]

RULE 13: RECURSION I

\[
\{ p_1 \} \, s_1.m_1(\bar{t}_1) \, \{ q_1 \}, \ldots, \{ p_n \} \, s_n.m_n(\bar{t}_n) \, \{ q_n \} \vdash \{ p \} \, S \, \{ q \},
\]

\[
\{ p_i \} \, \begin{local} \text{this}, \bar{u}_i := s_i, \bar{t}_i; S; \end{local} \, \{ q_i \}, \ i \in \{ 1, \ldots, n \}
\]

\[
\{ p \} \, S \, \{ q \}
\]

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

RULE 14: RECURSION II

\[
\{ p_1 \} \, s_1.m_1(\bar{t}_1) \, \{ q_1 \}, \ldots, \{ p_n \} \, s_n.m_n(\bar{t}_n) \, \{ q_n \} \vdash \{ p \} \, S \, \{ q \},
\]

\[
\{ p_i \} \, \begin{local} \text{this}, \bar{u}_i := s_i, \bar{t}_i; S; \end{local} \, \{ q_i \}, \ i \in \{ 1, \ldots, n \}
\]

\[
p_i \to s_i \neq \text{null}, \ i \in \{ 1, \ldots, n \}
\]

\[
\{ p \} \, S \, \{ q \}
\]

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**

\[
\begin{align*}
\{p_1\} P_1(t_1) \{q_1\}, \ldots, \{p_n\} P_n(t_n) \{q_n\} \vdash \{p\} S \{q\}, \\
\{p_1\} P_1(t_1) \{q_1\}, \ldots, \{p_n\} P_n(t_n) \{q_n\} \vdash \\
\{p_i\} \text{begin local } \bar{u}_i := t_i; S_i \text{ end } \{q_i\}, \ i \in \{1, \ldots, n\} \\
\{p\} S \{q\}
\end{align*}
\]

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

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