Completeness and complexity of reasoning about call-by-value in Hoare logic

Boer, F.S. de; Hiep, H.A.

Completeness and Complexity of Reasoning about Call-by-Value in Hoare Logic

FRANK S. DE BOER and HANS-DIETER A. HIEP, University of Leiden, The Netherlands and CWI, The Netherlands

We provide a sound and relatively complete Hoare logic for reasoning about partial correctness of recursive procedures in presence of local variables and the call-by-value parameter mechanism and in which the correctness proofs support contracts and are linear in the length of the program. We argue that in spite of the fact that Hoare logics for recursive procedures were intensively studied, no such logic has been proposed in the literature.

CCS Concepts: • **Theory of computation → Semantics and reasoning;**

Additional Key Words and Phrases: Recursive procedures, call-by-value, Hoare logic, completeness, soundness

ACM Reference format:
Frank S. de Boer and Hans-Dieter A. Hiep. 2021. Completeness and Complexity of Reasoning about Call-by-Value in Hoare Logic. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 17 (October 2021), 35 pages.
https://doi.org/10.1145/3477143

1 INTRODUCTION

1.1 Background and Motivation

Hoare logic was originally introduced by C. A. R. Hoare [23]. It is the most widely used approach to program verification. Since the early 1970s, it has been successfully extended to several classes of programs, including parallel and object-oriented ones, see, e.g., the textbook [4]. The recent survey on Hoare logic [6] by K. R. Apt and E.-R. Olderog traces these historical developments. Also, formalization of Hoare logic in various interactive theorem provers, for example Coq (see, e.g., References [9, 31]), led to a computer aided verification of several programming languages.

One of the crucial features of Hoare logic is its syntax-oriented style. It makes it possible to formally justify annotations of programs at relevant places (for example at the entrance of each loop) with *invariants*. Such annotations crucially increase programmer's confidence in the correctness of the program. In turn, intended behaviour of procedures can be described by means of pre- and postconditions, which simplifies program development and supports the crucial "Design by Contract" methodology [26] employed by many large-scale software artifacts. The basic idea of this methodology is that a pre- and postcondition together form a contract between the user (client) of a procedure and the procedure itself: If the user ensures that whenever the procedure is used

Authors' address: F. S. de Boer and H.-D. A. Hiep, Centrum Wiskunde & Informatica (CWI), P.O. Box 94079, 1090 GB, Amsterdam, The Netherlands and Leiden Institute of Advanced Computer Science (LIACS), University of Leiden, P.O. Box 9512, 2300 RA, Leiden, The Netherlands; emails: {frb, hdh}@cwi.nl.

0164-0925/2021/10-ART17 $15.00
https://doi.org/10.1145/3477143

(called) the precondition holds, then the procedure ensures the corresponding postcondition. In this manner, correct use of procedures does not require knowledge of the implementation details, which is of crucial importance in mastering both the complexity of software development and software verification. For example, one of the state-of-the-art theorem provers for the verification of Java programs, the KeY system [1], is based on the Java Modeling Language [7] that supports the specification of contracts for method definitions in Java. Instead of verifying the correctness of a procedure (or method) definition for each call separately, contracts are verified only once, which reduces the total verification effort (for example, in the verification of OpenJDK's `sort` method for generic collections, see Reference [16]).

The developments that further motivate the subject of the present article passed through a number of crucial stages. Already in Reference [24] a proof system for recursive procedures with parameters was proposed by Hoare that was subsequently used in Reference [18], where Hoare and M. Foley establish the correctness of the `Quicksort` program. This research was furthered by S. A. Cook [11], who proposed a by-now-standard notion of *relative completeness*, since then called *completeness in the sense of Cook*, and established relative completeness of a proof system for non-recursive procedures. Cook's result was extended by G. A. Gorelick [21], where a proof system for recursive procedures was introduced and proved to be sound and relatively complete. This line of research led to the seminal paper of E. M. Clarke [10], who exhibited a combination of five programming features, the presence of which makes it impossible to obtain a Hoare-like proof system that is sound and relatively complete.

However, often overlooked is that all these papers assumed the by now obsolete call-by-name parameter mechanism. Our claim is that no paper so far provided a sound and relatively complete Hoare-like proof system for a programming language with the following programming features:

- a system of mutually recursive procedures with explicit parameter declarations,
- global and local variables,
- call-by-value parameter mechanism,
- both dynamic and static scope,[1]

and in which

- reasoning about *contracts* is supported, and thus,
- correctness proofs are *linear* in the length of the programs.

Given the above research and the fact that in many programming languages call-by-value is the main parameter mechanism, this claim may sound surprising. Of course, there were several contributions to Hoare logic concerned with recursive procedures, but none of them provided a proof system that met the above criteria. The aim of this article is to provide such a Hoare-like proof system and thus establish a link between the practice and theory of software verification by a formal justification of the use of contracts in the verification of mainstream programming languages like Java.

1.2 Related Work

The origin of recursive procedures in the context of programming languages is studied in Reference [33].

The first sound and relatively complete proof system for programs with local variables and recursive procedures was provided in the thesis of Gorelick [21]. But that paper assumed the call-by-name parameter mechanism and, as explained in Reference [3, pp. 459–460], dynamic scope was assumed. The relative completeness result also assumed some restrictions on the actual parameters

[1] Both notions are explained in Section 2. Almost all programming languages assume static scope.

in the procedure calls that were partly lifted by R. Cartwright and D. C. Oppen [8]. However, the restriction that global variables do not occur in the actual parameters is still present there.

In Reference [14] and in more detail in Reference [15, Section 9.4], J. W. de Bakker proposed a proof system concerned with the recursive procedures with the call-by-value and call-by-variable (present in Pascal) parameter mechanisms in presence of static scope and its proof of soundness and relative completeness. However, the correctness of each procedure call had to be proved separately, that is, each procedure call requires a separate correctness proof of the corresponding procedure body. As a result correctness proofs are only quadratic in the length of the programs, even in the presence of just one recursive procedure. As already stated above, in practice, however, procedures (or methods in object-oriented languages) are specified by contracts that provide a generic specification for each procedure call and allow for correctness proofs that are linear in the length of the programs.

Further, the relative completeness was established only for the special case of a single recursive procedure with a single recursive call. For the case of two recursive calls a list of 14 cases was presented in Reference [15, Section 9.4] that should be considered, but without the corresponding proofs. The main ideas of this proof were discussed in Reference [3]. The case of a larger number of recursive calls and a system of mutually recursive procedures were not analyzed because of the complications resulting from an accumulation of cases generated by the use of several variable substitutions.

In previous work [4, 5] (both co-authored by F. S. de Boer), a proof system was proposed for the programming language here considered and shown to be sound and relatively complete. However, also here correctness of each procedure call has to be dealt with separately, even in the presence of just one recursive procedure. An attempt was made to circumvent this inefficiency by proposing a proof rule for the *instantiation* of contracts [4, pp. 159–160] by replacing the formal parameters in both the pre- and postcondition of the contract by the actual parameters. However, this requires, among others, that the variables appearing in the actual parameters cannot be changed by the execution of the call. This is rather restrictive in practice (see Example 4.3 of Section 4).

D. von Oheimb [34] discusses a sound and relatively complete proof system for a programming language with mutually recursive procedures and a call-by-value parameter mechanism. The proofs were certified in the Isabelle theorem prover instantiated with Church's higher-order logic (Isabelle/HOL). The formalization of the proof system itself abstracts from the syntactical representation of assertions, as stated as follows in Reference [34]:

> Central to any axiomatic semantics is the notion of assertions, which describe properties of the program state before and after executing commands. Semantically speaking, assertions are just predicates on the state. We adopt this abstract view (similarly to our semantic view of expressions) and thus avoid talking explicitly on a syntactic level about terms and substitution and their interpretation.

Usually, in Hoare logic the formal language of *predicate logic* is used to represent syntactically the semantic relations between the values of the program variables. One of the main challenges of designing a Hoare-like logic is then to formalize the semantics of programs declaratively in predicate logic, *at an abstraction level that coincides with that of the programming language*. The abstract view of assertions blurs the clear distinction between the declarative nature of assertions and the operational semantics of programs. For example, in Reference [34] the assertions used in the proof rules for block statements and recursive procedure calls assume operations that are used to define *operationally* the program semantics, but that are not part of the programming language itself. Further, in Reference [34] assertions assume a program-independent distinction between local and global variables. In our work, we use assertions in the formal language of predicate logic. As such, we make no program-independent distinction between local and global variables within the assertion language: There is only the syntactical notion of free and bound variable occurrences. Further, our work allows the assertion language to have as interpretation an *arbitrary* underlying structure, e.g., the integers and Booleans. Thus, we are limited in how to formalize proof rules for dealing with local variables and parameters. This leaves us with the only logical possibilities of either restricting variable occurrences, or by renaming variables, in assertions.

We conclude that the relative completeness result presented here is new. It is useful to discuss how we dealt with the complications encountered in the reported papers.

One of the notorious problems when dealing with the above programming features is that variables can occur in a program both as local and global. The way this double use of variables is dealt with has direct consequences on what is being formalized. Additionally, variables can be used as formal parameters and can occur in actual parameters. This multiple use of variables can lead to various subtle errors and was usually dealt with by imposing some restrictions on the actual parameters, notably that the variables of the actual parameters cannot be changed by the call. In our approach no such restrictions are present but these complications explain the seemingly excessive care we exercise when dealing with this matter.

In the relative completeness proofs of Cook [11], Gorelick [21], and De Bakker [15], the main difficulties had to do with the local variables, the use of which led to some variable renamings, and the clashes between various types of variables, for example formal parameters and global variables, that led to some syntactic restrictions.

In fact, the original paper of Cook [11] contains an error that was corrected in Reference [12]. It is useful to discuss it (it was actually pointed out by Apt) in some detail. In the semantics adopted in Reference [11] local variables were modelled using a stack in which the last used value was kept on the stack and implicitly assigned to the next local variable. As a result, we have that $y = 1$ holds after

$$\textbf{begin local } x;\ x := 1 \textbf{ end};\ \textbf{begin local } x;\ y := x \textbf{ end}$$

is executed. However, there is no way to prove it. In Gorelick's thesis [21] this error does not arise, since all local variables are explicitly initialized to a given-in-advance value, both in the semantics and in the proof theory (by adjusting the precondition of the corresponding declaration rule).

However, this problem does arise in the framework of Cartwright and Oppen [8], where the authors write on p. 371:

> As Apt (personal communication) has observed, this rule [for local variables] is incomplete because it does not allow one to deduce the values of new variables on block entry. There are several possible solutions to this technical problem but they are beyond the scope of this paper.

In our framework this problem cannot occur because of the explicit *initialization* of the local variables in the block statement. However, our initialization is more general than that of Gorelick. For example, in the statement **begin local** $u := u;\ S$ **end** the *local* variable u is initialized to the value of the *global* variable u. Consequently, according to our semantics, and also the semantics used in Reference [11], $x = y$ holds after execution of the statement

$$\textbf{begin local } u := u;\ x := u \textbf{ end};\ \textbf{begin local } u := u;\ y := u \textbf{ end}.$$

This cannot be proved in the proof system used in Reference [11]. However, we can prove it in our proof system (as shown in Example 4.2).

1.3 Summary of Our Approach

We prove relative completeness for our programming language that allows for dynamic scoping of local variables. However, for a natural syntactically defined subset of programs that avoid name clashes between global and local variables, static scoping of local variables is ensured. By the general nature of our completeness result (which holds for *all* programs), we also obtain relative completeness for this sub-language. This class of programs was first considered in Reference [8], where it is introduced on p. 372 in a somewhat informal way:

> [...] we assume that our PASCAL subset [...] requires that the global variables accessed by a procedure be explicitly declared at the head of the procedure and that these variables be accessible at the point of every call.

Let us discuss now how we succeeded to circumvent the complications reported above. We achieved it by various design decisions concerning the syntax and semantics, which resulted in a simple proof system. Some of these decisions were already taken in the textbook [4] and used in Reference [5] (both co-authored by F. S. de Boer). More precisely, the block statement that declares local variables must include explicit expressions used for initialization. This, in conjunction with the parallel assignment, allows one to model procedure calls in a simple way, by inlining.

Crucially, the semantics of such block statements does not require any *variable renaming*. This leads to a simple semantics of the procedure calls, without any variable renaming either.

As a result, in contrast to all other works in the literature, our BLOCK rule dealing with local variables uses no substitution. In contrast, in Reference [24] the substitution is applied to the program, while in Reference [11] and Reference [21] it is applied to the assertions. Further, in contrast to Reference [15], our RECURSION rule does not involve any substitution in the procedure body. This allowed us to circumvent the troublesome combinatorial explosion of the cases encountered in the relative completeness proof of Reference [15].

However, the key improvement over previous work [4, 5], is that, here, the RECURSION rule formalizes reasoning about contracts and, crucially, the PROCEDURE CALL rule does not impose any restrictions on the actual parameters. The latter is in contrast to all works that dealt with the call-by-name parameter mechanism. Thanks to this improvement, in contrast to the above two works, in our proof system the correctness proofs support contracts and as such are linear in the length of the program.

1.4 Plan of the Paper

In the next section, we introduce a programming language that includes all the basic features of recursive call-by-value procedures and identify a natural subset of clash-free programs for which dynamic and static scope coincide. Next, in Section 3, we recall various aspects of semantics introduced in the textbook [4] and establish some properties that are used when reasoning about the considered proof system. Section 4 introduces and motivates the design of a proof system for reasoning about recursive calls by means of inlining (or body replacement) and show how we can do better by means of contracts. Section 5 discusses the complexity of proofs and a proof normalization procedure for obtaining linear proofs. In Section 6, we show that the proof system given in Section 4 is sound. In Section 7, we establish relative completeness of the proof system. In Section 8, we describe a formalization of the main semantic argument underlying the relative completeness result in the theorem prover Coq. Finally, in Section 9 we discuss future work and conclude the article.

2 SYNTAX

Throughout the article, we assume a fixed first-order language $\mathcal{L}$. *Expressions* are terms in the language $\mathcal{L}$, *Boolean expressions* are quantifier-free formulas of $\mathcal{L}$, while *assertions* are formulas of $\mathcal{L}$, which are considered equal up to alphabetic renaming of quantified variables.

We denote the set of all variables of $\mathcal{L}$ by *Var*. For a sequence $\bar{x}$ of distinct variables, we denote by $\{\bar{x}\}$ its corresponding set. For a Boolean expression B or a sequence $\bar{t}$ of expressions, we denote the set of all variables occurring in B or in $\bar{t}$ by, respectively, $var(B)$ and $var(\bar{t})$. The set of variables that occur free in an assertion p is defined in a standard way and denoted by $free(p)$.

A (simultaneous) *substitution* of a list of expressions $\bar{t}$ for a list of distinct variables $\bar{x}$ of the same length is written as $[\bar{x} := \bar{t}]$ and the result of applying it to an expression or an assertion s as $s[\bar{x} := \bar{t}]$. To ensure a uniform presentation, we allow the empty substitution in which the list of variables $\bar{x}$ is empty.

We now move on and introduce the syntax of the programs. For simplicity in the considered programming language, we admit only simple variables (so no array or subscripted variables), all of the same type. *Statements* are defined by the following grammar:

$$\begin{aligned} S \quad ::= \quad & \textbf{skip} \mid \bar{x} := \bar{t} \mid P(\bar{t}) \mid S;\ S \mid \textbf{if } B \textbf{ then } S \textbf{ else } S \textbf{ fi} \mid \\ & \textbf{while } B \textbf{ do } S \textbf{ od} \mid \textbf{begin local } \bar{x} := \bar{t};\ S \textbf{ end}, \end{aligned}$$

where

- **skip** denotes the “empty” statement,
- $\bar{x} := \bar{t}$ is a *parallel assignment*, with $\bar{x}$ a (possibly empty) list of distinct variables and $\bar{t}$ a list of expressions of the same length as $\bar{x}$; when $\bar{x}$ is empty, we identify $\bar{x} := \bar{t}$ with the **skip** statement,
- P is a procedure name; each procedure P is defined by a declaration of the form

$$P(\bar{u}) :: S,$$

  where $\bar{u}$ is a (possibly empty) list of distinct variables, called *formal parameters* of the procedure P, and S is a statement, called the *body* of the procedure P,
- $P(\bar{t})$ is a procedure call, with the *actual parameters* $\bar{t}$, which is a (possibly empty) list of expressions of the same length as the corresponding list of formal parameters,
- B is a Boolean expression,
- **begin local** $\bar{x} := \bar{t};\ S$ **end** is a *block statement* where $\bar{x}$ is a (possibly empty) list of distinct local variables, all of which are explicitly initialized by means of the parallel assignment $\bar{x} := \bar{t}$.

By a *program*, we mean a pair $(D \mid S)$, where S is a statement, called the *main statement* and D is a set of procedure declarations such that each procedure (name) that appears in S or D has exactly one procedure declaration in D. So we allow mutually recursive procedures but not nested procedures. We denote by $var(D \mid S)$ the set of variables that occur in $(D \mid S)$.

By a *correctness formula*, we mean a triple $\{p\}\ S\ \{q\}$, where p, q are assertions and S is a statement, or a triple $\{p\}\ D \mid S\ \{q\}$, where $(D \mid S)$ is a program. Of special interest in our approach will be the representation of *contracts* that specify by means of a precondition and postcondition the input/output behavior of a procedure. We represent them by the correctness formulas $\{p\}\ P(\bar{u})\ \{q\}$, where $\bar{u}$ are the *formal parameters* of P. Such a call $P(\bar{u})$ is called a *generic* procedure call.[2] Note that so defined contracts abstract from the implementation of the specified procedure.

The parallel assignment plays a crucial role in the way procedure calls are dealt with: the procedure call $P(\bar{t})$, where P is declared by $P(\bar{u}) :: S$, is interpreted as the block statement **begin local** $\bar{u} := \bar{t};\ S$ **end**, where $\bar{u} := \bar{t}$ models the parameter passing by value and the block statement ensures that the changes to the formal parameters $\bar{u}$ are local. Such a replacement of a procedure call by an appropriately modified procedure body is called *inlining* or a *copy rule*.

[2] Not to be confused with generic types (e.g., as in Java) of the formal parameters. Our basic programming language does *not* have multiple different types, nor generic types.

In our setup inlining results in a *dynamic scope* of local variables so that each procedure call is evaluated in the environment in which it is called. The simplest example is the parameterless procedure P declared by $P() :: y := x$ and the main statement $x := 0;$ **begin local** $x := 1;\ P()$ **end**. Here the inlining results in the program

$$x := 0;\ \textbf{begin local}\ x := 1;\ \textbf{begin local skip};\ y := x\ \textbf{end end}$$

that yields $y = 1$ upon termination. However, if we renamed the occurrence of x in the block statement to a fresh variable, say, x', and thus used the statement $x := 0;$ **begin local** $x' := 1;\ P()$ **end**, then inlining would result in the program

$$x := 0;\ \textbf{begin local}\ x' := 1;\ \textbf{begin local skip};\ y := x\ \textbf{end end}$$

that yields $y = 0$ upon termination. Thus renaming of local variables (e.g., also known as alpha conversion in the lambda calculus) affects the semantics. However, renaming of local variables upon inlining, to avoid clashes between global and local variables, gives rise to a *static scope* of local variables, which ensures that each procedure call is evaluated in the environment in which it is declared.

The above example shows that static scope can also be ensured when certain variable name clashes are avoided. This can be made precise as follows.

Definition 2.1 (Global Variables). Given a program $(D \mid S)$, we define the set of *global* variables $Gvar(D \mid S)$ as follows.

- $Gvar(D \mid \textbf{skip}) = \emptyset$,
- $Gvar(D \mid \bar{x} := \bar{t}) = \{\bar{x}\} \cup var(\bar{t})$,
- $Gvar(D \mid P(\bar{t})) = Gvar(D \mid \textbf{begin local}\ \bar{u} := \bar{t};\ S\ \textbf{end})$ where $P(\bar{u}) :: S \in D$,
- $Gvar(D \mid S_1;\ S_2) = Gvar(D \mid S_1) \cup Gvar(D \mid S_2)$,
- $Gvar(D \mid \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}) = var(B) \cup Gvar(D \mid S_1) \cup Gvar(D \mid S_2)$,
- $Gvar(D \mid \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}) = var(B) \cup Gvar(D \mid S)$,
- $Gvar(D \mid \textbf{begin local}\ \bar{x} := \bar{t};\ S\ \textbf{end}) = var(\bar{t}) \cup (Gvar(D \mid S) \setminus \{\bar{x}\})$,

Finally, we define

- $Gvar(D \mid P) = Gvar(D \mid S) \setminus \{\bar{u}\}$, where $P(\bar{u}) :: S \in D$.

To formally justify the circularity caused by the presence of recursive calls, we use a sequence of $Gvar^k$ functions, where $k \geq 0$, defined as follows. For each of them, we reuse the above equalities as definitions, except for the case of the procedure calls for which we set

- $Gvar^0(D \mid P(\bar{t})) = \emptyset$,
- $Gvar^{k+1}(D \mid P(\bar{t})) = Gvar^k(D \mid \textbf{begin local}\ \bar{u} := \bar{t};\ S\ \textbf{end})$,

where $P(\bar{u}) :: S \in D$. A simple proof by induction shows that for all $k \geq 0$

$$var(D \mid S) \supseteq Gvar^{k+1}(D \mid S) \supseteq Gvar^k(D \mid S),$$

so for each procedure call $P(\bar{t})$

$$\exists k \geq 0\ \forall l \geq k\ Gvar^{l+1}(D \mid P(\bar{t})) = Gvar^l(D \mid P(\bar{t})).$$

Denote the least such k by $k(P(\bar{t}))$. We identify then $Gvar(D \mid S)$ with $Gvar^l(D \mid S)$, where $l = \max\{k(P(\bar{t})) \mid P(\bar{t})\ \text{occurs in}\ (D \mid S)\} + 1$. Clearly, $Gvar(D \mid S)$ thus defined satisfies the original equalities. □

It is important to note that in the statement **begin local** $\bar{x} := \bar{t};\ S$ **end** the variables occurring in $\bar{t}$ are global, and as such those variables $\bar{x}$ appearing in $\bar{t}$ are interpreted differently from the local ones that only exist in the context of the block statement (as formalized in Section 3). We view the variables from $Gvar(D \mid P)$ as the global variables of the procedure P.

Definition 2.2 (Lexical Scope). A variable $y \in Gvar(D \mid P)$ appears in the (lexical) *scope* of a block statement **begin local** $\bar{x} := \bar{t};\ S$ **end**, if y appears in the list of local variables $\bar{x}$ and S contains a call of P. A program $(D \mid S)$ is called *clash-free* if the following holds for every procedure P declared in D: no variable $y \in Gvar(D \mid P)$ appears in the scope of a block statement that occurs in $(D \mid S)$.

Clash-free programs by definition thus ensure static scoping of local variables. So a programmer in the considered programming language can ensure static scope by adhering to a simple syntactic convention. This syntactic approach to static scoping also avoids the need for a different semantics of the block statement and allows for an uniform proof theory.

In our considerations it will be important to refer to the set of variables that a given program *may* change (but not necessarily *does* change).

Definition 2.3 (Write Variables). Given a program $(D \mid S)$, we define the set of variables that may be changed by it as follows. Consider the following equalities:

- $change(D \mid \textbf{skip}) = \emptyset$,
- $change(D \mid \bar{x} := \bar{t}) = \{\bar{x}\}$,
- $change(D \mid P(\bar{t})) = change(D \mid \textbf{begin local}\ \bar{u} := \bar{t};\ S\ \textbf{end})$, where $P(\bar{u}) :: S \in D$,
- $change(D \mid S_1;\ S_2) = change(D \mid S_1) \cup change(D \mid S_2)$,
- $change(D \mid \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}) = change(D \mid S_1) \cup change(D \mid S_2)$,
- $change(D \mid \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}) = change(D \mid S)$,
- $change(D \mid \textbf{begin local}\ \bar{x} := \bar{t};\ S\ \textbf{end}) = change(D \mid S) \setminus \{\bar{x}\}$.

The circularity of this definition can be resolved as above. □

3 SEMANTICS

In this section, we gather various basic facts concerning semantics of our programming language. We begin by a slightly adjusted presentation extracted from the textbook [4], followed by a collection of various properties that will be needed later.

As already mentioned, we assume for simplicity that all variables are of the same type, say, T. Each realization of this type (called a *domain)* yields an *interpretation* I that assigns a meaning to the function symbols and relation symbols of the language $\mathcal{L}$.

Throughout this article, we assume a fixed interpretation I. By a *state*, we mean a function that maps each variable to an element of the domain of I. We denote the set of states by Σ.

An *update* $\sigma[x := d]$ of a state σ, where x is a variable and d an element of the domain of I, is a state that coincides with σ on all variables except x to which it assigns d. A *simultaneous* update $\sigma[\bar{x} := \bar{d}]$ of a state σ, where $\bar{x}$ is a list of (distinct) variables and $\bar{d}$ is a list of domain elements of the same length, is defined analogously.

Given a state σ and an expression t, we define the element $\sigma(t)$ of the domain of I inductively in the standard way and for $\bar{t} := t_1, \ldots, t_k$ we put $\sigma(\bar{t}) := (\sigma(t_1), \ldots, \sigma(t_k))$.

For a set Z of variables, we denote by $\sigma[Z]$ the *restriction* of the state σ to the variables occurring in Z and write for two states σ and τ

$$\sigma = \tau\ \textbf{mod}\ Z$$

if $\sigma[Var \backslash Z] = \tau[Var \backslash Z]$. We extend the definitions of an update and equality **mod** Z to, respectively, a set of states and sets of states in the expected way. So for a set of states X

$$X[x := d] = \{\sigma[x := d] \mid \sigma \in X\}$$

By $(\sigma, D \mid S) \Rightarrow \sigma'$ we denote that the execution of program $(D \mid S)$ starting in state σ terminates in state σ', and is defined as the smallest relation satisfying:

$$(\sigma, D \mid \textbf{skip}) \Rightarrow \sigma \qquad (\sigma, D \mid \bar{x} := \bar{t}) \Rightarrow \sigma[\bar{x} := \sigma(\bar{t})]$$

$$\frac{(\sigma, D \mid S_1) \Rightarrow \sigma' \quad (\sigma', D \mid S_2) \Rightarrow \sigma''}{(\sigma, D \mid S_1;\ S_2) \Rightarrow \sigma''}$$

$$\frac{(\sigma, D \mid \bar{x} := \bar{t};\ S) \Rightarrow \sigma'}{(\sigma, D \mid \textbf{begin local } \bar{x} := \bar{t};\ S \textbf{ end}) \Rightarrow \sigma'[\bar{x} := \sigma(\bar{x})]}$$

$$\frac{(\sigma, D \mid \textbf{begin local } \bar{u} := \bar{t};\ S \textbf{ end}) \Rightarrow \sigma' \quad P(\bar{u}) :: S \in D}{(\sigma, D \mid P(\bar{t})) \Rightarrow \sigma'}$$

$$\frac{\sigma \models B \quad (\sigma, D \mid S_1) \Rightarrow \sigma'}{(\sigma, D \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) \Rightarrow \sigma'} \qquad \frac{\sigma \not\models B \quad (\sigma, D \mid S_2) \Rightarrow \sigma'}{(\sigma, D \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) \Rightarrow \sigma'}$$

$$\frac{\sigma \models B \quad (\sigma, D \mid S;\ \textbf{while } B \textbf{ do } S \textbf{ od}) \Rightarrow \sigma'}{(\sigma, D \mid \textbf{while } B \textbf{ do } S \textbf{ od}) \Rightarrow \sigma'} \qquad \frac{\sigma \not\models B}{(\sigma, D \mid \textbf{while } B \textbf{ do } S \textbf{ od}) \Rightarrow \sigma}$$

Further, $\mathcal{M}[\![D \mid S]\!] : \Sigma \to \mathcal{P}(\Sigma)$ denotes the function:

$$\mathcal{M}[\![D \mid S]\!](\sigma) = \{\sigma' \mid (\sigma, D \mid S) \Rightarrow \sigma'\}.$$

Fig. 1. Big-step operational semantics.

and for two sets of states X and Y and a set of variables Z,

$$X = Y \textbf{ mod } Z \text{ if } \{\sigma[Var \setminus Z] \mid \sigma \in X\} = \{\sigma[Var \setminus Z] \mid \sigma \in Y\}.$$

The relation "assertion p is true in the state $\sigma \in \Sigma$," denoted by $\sigma \models p$, is defined inductively in the standard way. The *meaning* of an assertion (with respect to the interpretation I), written as $[\![p]\!]$, is defined by $[\![p]\!] = \{\sigma \in \Sigma \mid \sigma \models p\}$. We say that p is *valid*, and write $\models p$, if $[\![p]\!] = \Sigma$.

The *meaning* of a program $(D \mid S)$ is a function $\mathcal{M}[\![D \mid S]\!] : \Sigma \to \mathcal{P}(\Sigma)$ of which the small-step operational semantics is given in the textbook [4]. See Figure 1 for its big-step operational semantics. The operational definition of the assignment, sequential composition, choice and iteration constructs are standard. A procedure call is defined in terms of a corresponding block statement. The semantics of a block statement **begin local** $\bar{x} := \bar{t};\ S$ **end** *resets* the local variables $\bar{x}$ after the execution of the statement $\bar{x} := \bar{t};\ S$. Note that since $\bar{t}$ are evaluated in the *initial* state, the variables $\bar{x}$ appearing in $\bar{t}$ also refer to their values in the initial state.

For a given state σ, $\mathcal{M}[\![D \mid S]\!](\sigma) = \{\tau\}$ states the fact that the program $(D \mid S)$ terminates when started in the initial state σ, yielding the final state τ. If $(D \mid S)$ does not terminate when started in σ, then $\mathcal{M}[\![D \mid S]\!](\sigma)$ is the empty set.

We extend the function $\mathcal{M}[\![D \mid S]\!]$ to deal with sets of states $X \subseteq \Sigma$ by

$$\mathcal{M}[\![D \mid S]\!](X) = \bigcup_{\sigma \in X} \mathcal{M}[\![D \mid S]\!](\sigma).$$

Definition 3.1 (Syntactic Approximation). Given $D = \{P_1(\bar{u}_1) :: S_1, \ldots, P_n(\bar{u}_n) :: S_n\}$ and a statement S, we define the *kth syntactic approximation* S^k of S by induction on $k \geq 0$:

$$\begin{array}{lcl} S^0 & = & \Omega, \\ S^{k+1} & = & S[S_1^k/P_1, \ldots, S_n^k/P_n], \end{array}$$

where Ω denotes a diverging statement (e.g., **while true do skip od**) and $S[R_1/P_1, \ldots, R_n/P_n]$ is the result of a simultaneous replacement in S of each procedure call $P_i(\bar{t})$ by

$$\textbf{begin local } \bar{u}_i := \bar{t};\ R_i \textbf{ end}.$$

Furthermore, let D^k stand for $\{P_1(\bar{u}_1) :: S_1^k, \ldots, P_n(\bar{u}_n) :: S_n^k\}$.

Lemma 3.2 (Basic Semantic Properties).

Skip *For all states* σ

$$\mathcal{M}[\![D \mid \textbf{skip}]\!](\sigma) = \{\sigma\}.$$

Assignment *For all states* σ

$$\mathcal{M}[\![D \mid \bar{x} := \bar{t}]\!](\sigma) = \{\sigma[\bar{x} := \sigma(\bar{t})]\}.$$

Composition *For all states* σ

$$\mathcal{M}[\![D \mid S_1;\ S_2]\!](\sigma) = \mathcal{M}[\![D \mid S_2]\!](\mathcal{M}[\![D \mid S_1]\!](\sigma)).$$

Block *For all states* σ *and where* $X = \mathcal{M}[\![D \mid \bar{x} := \bar{t};\ S]\!](\sigma)$,

$$\mathcal{M}[\![D \mid \textbf{begin local } \bar{x} := \bar{t};\ S \textbf{ end}]\!](\sigma) = X[\bar{x} := \sigma(\bar{x})].$$

Inlining *For* $D = \{P_1(\bar{u}_1) :: S_1, \ldots, P_n(\bar{u}_n) :: S_n\}$

$$\mathcal{M}[\![D \mid S]\!] = \mathcal{M}[\![D \mid S[S_1/P_1, \ldots, S_n/P_n]]\!].$$

As a special case, we have for any procedure P *declared by* $P(\bar{u}) :: S \in D$,

$$\mathcal{M}[\![D \mid P(\bar{t})]\!] = \mathcal{M}[\![D \mid \textbf{begin local } \bar{u} := \bar{t};\ S \textbf{ end}]\!].$$

Approximation

$$\mathcal{M}[\![D \mid S]\!] = \bigcup_{k=0}^{\infty} \mathcal{M}[\![D^k \mid S]\!].$$

Access and Change *For all states* σ *and* τ *if* $\mathcal{M}[\![D \mid S]\!](\sigma) = \{\tau\}$ *then*

$$\tau[\mathit{Var} \setminus \mathit{change}(D \mid S)] = \sigma[\mathit{Var} \setminus \mathit{change}(D \mid S)].$$

Lemma 3.2 collects various basic properties of the big-step operational semantics (their proofs are standard, and therefore omitted). Note that the clauses for the basic constructs can also be interpreted independently as a *Definition* of a *denotational* semantics.

The **Block** item employs the earlier introduced extension of the update operation to a set of states and is a concise way of writing that for all states σ

$$\mathcal{M}[\![D \mid \textbf{begin local } \bar{x} := \bar{t};\ S \textbf{ end}]\!](\sigma) = \{\sigma'[\bar{x} := \sigma(\bar{x})] \mid \sigma' \in \mathcal{M}[\![D \mid \bar{x} := \bar{t};\ S]\!](\sigma)\}.$$

It states that the execution of a block statement consists of the initialization of the local variables, followed by the execution of its body, and a subsequent reset of the local variables to their initial values. The **Access and Change** item formalizes the intuition that when executing a program $(D \mid S)$ only variables in $\mathit{change}(D \mid S)$ can be modified. It can be equivalently stated as that $\mathcal{M}[\![D \mid S]\!](\sigma) \neq \emptyset$ implies

$$\mathcal{M}[\![D \mid S]\!](\sigma) = \{\sigma\} \textbf{ mod } \mathit{change}(D \mid S).$$

Combining the above **Block** and **Inlining** items, we obtain that the execution of a procedure call consists of the initialization of the formal parameters to the actual parameters, followed by the execution of the procedure body, and a subsequent reset of the formal parameters to their initial values. This is stated by following corollary.

Corollary 3.3. *Suppose that the procedure P is declared by $P(\bar{u}) :: S$. For all states σ*

$$\mathcal{M}[\![D \mid P(\bar{t})]\!](\sigma) = X[\bar{u} := \sigma(\bar{u})],$$

where $X = \mathcal{M}[\![D \mid S]\!](\sigma[\bar{u} := \sigma(\bar{t})]) = \mathcal{M}[\![D \mid \bar{u} := \bar{t}; S]\!](\sigma)$.

Proof. By the appropriate items of Lemma 3.2, we successively have

$$\begin{array}{rl} & \mathcal{M}[\![D \mid P(\bar{t})]\!](\sigma) \\ = & \mathcal{M}[\![D \mid \textbf{begin local } \bar{u} := \bar{t};\ S \textbf{ end}]\!](\sigma) \\ = & \big(\mathcal{M}[\![D \mid \bar{u} := \bar{t};\ S]\!](\sigma)\big)[\bar{u} := \sigma(\bar{u})] \\ = & \big(\mathcal{M}[\![D \mid S]\!](\sigma[\bar{u} := \sigma(\bar{t})])\big)[\bar{u} := \sigma(\bar{u})]. \end{array}$$ □

Definition 3.4 (Semantics Correctness Formulas). Given a program $(D \mid S)$ and a correctness formula $\{p\}\ D \mid S\ \{q\}$, we write

$$\models \{p\}\ D \mid S\ \{q\}$$

if

$$\mathcal{M}[\![D \mid S]\!]([\![p]\!]) \subseteq [\![q]\!].$$

We say then that $\{p\}\ D \mid S\ \{q\}$ is true *in the sense of partial correctness.*

The following lemma states a basic property of the semantics of correctness formulas.

Lemma 3.5. *Suppose that for all states σ*

$$\mathcal{M}[\![D \mid S]\!](\sigma) = \mathcal{M}[\![D \mid T]\!](\sigma) \textbf{ mod } \{\bar{u}\}.$$

Then for all assertions p and q such that $\{\bar{u}\} \cap free(q) = \emptyset$

$$\models \{p\}\ D \mid S\ \{q\} \textit{ iff} \models \{p\}\ D \mid T\ \{q\}.$$

Proof. By the first assumption

$$\mathcal{M}[\![D \mid S]\!]([\![p]\!]) = \mathcal{M}[\![D \mid T]\!]([\![p]\!]) \textbf{ mod } \{\bar{u}\}. \tag{1}$$

By the second assumption $free(q) \subseteq Var \setminus \{\bar{u}\}$, so for arbitrary states σ and τ such that $\sigma = \tau \textbf{ mod } \{\bar{u}\}$ we have $\sigma \models q$ iff $\tau \models q$. Hence for two sets of states X and Y such that $X = Y \textbf{ mod } \{\bar{u}\}$, we have

$$X \subseteq [\![q]\!] \text{ iff } Y \subseteq [\![q]\!].$$

So the desired equivalence follows Equation (1) and the definition of $\models \{p\}\ D \mid S\ \{q\}$. □

Corollary 3.6. *For all assertions p and q such that $\{\bar{u}\} \cap free(q) = \emptyset$,*

(i) $\models \{p\}\ D \mid \textbf{begin local } \bar{u} := \bar{t};\ S \textbf{ end}\ \{q\}$ *iff* $\models \{p\}\ D \mid \bar{u} := \bar{t};\ S\ \{q\}$.
(ii) $\models \{p\}\ D \mid P(\bar{t})\ \{q\}$ *iff* $\models \{p\}\ D \mid \bar{u} := \bar{t};\ S\ \{q\}$,
where the procedure P is declared by $P(\bar{u}) :: S \in D$.

Proof. By the **Block** and **Inlining** items of Lemma 3.2,

$$\mathcal{M}[\![D \mid \textbf{begin local } \bar{u} := \bar{t};\ S \textbf{ end}]\!](\sigma) = \mathcal{M}[\![D \mid \bar{u} := \bar{t};\ S]\!](\sigma) \textbf{ mod } \{\bar{u}\}$$

and

$$\mathcal{M}[\![D \mid P(\bar{t})]\!](\sigma) = \mathcal{M}[\![D \mid \bar{u} := \bar{t};\ S]\!](\sigma) \textbf{ mod } \{\bar{u}\},$$

so the claim follows by Lemma 3.5. □

Following Reference [11], we now introduce the following crucial notions underlying the completeness proof.

Definition 3.7 (Strongest Postcondition). Denote by $SP(p, D \mid S)$ the *strongest postcondition* of the program $(D \mid S)$ with respect to an assertion p, defined by

$$SP(p, D \mid S) = \mathcal{M}[\![D \mid S]\!]([\![p]\!]).$$

So $SP(p, D \mid S)$ is the set of states that can be reached by executing $(D \mid S)$ starting in a state satisfying p.

We say that a set of states X is *definable* (in the given interpretation I) iff for some formula p of $\mathcal{L}$ we have $X = [\![p]\!]$. We say then that p *defines* Σ.

Definition 3.8 (Expressivity). The language $\mathcal{L}$ is *expressive* with respect to the given interpretation I, if for for every assertion p and program $(D \mid S)$ the set of states $SP(p, D \mid S)$ is *definable*.

Throughout this article, we assume that the language $\mathcal{L}$ is *expressive* with respect to the given interpretation I and identify $SP(p, D \mid S)$ with its defining formula.

Given a proof system for proving correctness formulas, we denote by

$$\vdash \{p\}\ D \mid S\ \{q\}$$

that $\{p\}\ D \mid S\ \{q\}$ can be proved by means of the proof system. We say that a proof system

- is *sound* if for every correctness formula $\{p\}\ D \mid S\ \{q\}$,
$$\vdash \{p\}\ D \mid S\ \{q\} \text{ implies } \models \{p\}\ D \mid S\ \{q\},$$
- is *complete*, if for every correctness formula $\{p\}\ D \mid S\ \{q\}$,
$$\models \{p\}\ D \mid S\ \{q\} \text{ implies } \vdash \{p\}\ D \mid S\ \{q\}.$$

4 PROOF SYSTEM

Figure 2 includes the usual basic Hoare logic for assignment and the basic control structures of sequential composition, choice and iteration (since the set of declarations is empty for this sublanguage, it is omitted). We use simultaneous substitution to generalize the standard assignment axiom. A skip statement is identified with an empty assignment, and thus we obtain as an instance of the (parallel) assignment axiom, the derived axiom $\{p\}$ **skip** $\{p\}$. Hoare actually introduced two consequence rules, that soon after were combined into the rule listed here. The proof rule concerning the **if-then-else** statement was originally proposed by P. Lauer [25].

We note here that by *collapsing* consecutive applications of the consequence rule into one, we can *normalize* any proof in our basic Hoare logic that consists of the consequence rule, the assignment axiom and the rules for the basic control structures of sequential composition, choice and iteration. Such a normalization gives rise to proofs that are linear in the size of the statement, measured by its number of constructs (assignments, sequential composition, choice and iteration). To be more precise, let $l(S)$ denote the number of constructs of S, inductively defined as follows.

Definition 4.1 (Size of a Statement). We inductively define the number $l(S)$ as follows:

- $l(\textbf{skip}) := 1$,
- $l(\bar{x} := \bar{t}) := 1$,
- $l(S_1;\ S_2) := l(S_1) + l(S_2) + 1$,
- $l(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) := l(S_1) + l(S_2) + 1$,
- $l(\textbf{while } B \textbf{ do } S \textbf{ od}) := l(S) + 1$,

It is easy to verify by induction of $l(S)$ that if $\{p\}\ S\ \{q\}$ is provable then there exists a proof that consists of at most $2 \times l(S)$ rule applications.

To prove a correctness formula $\{p\}\ D \mid S\ \{q\}$ about a recursive program $(D \mid S)$, we extend the basic Hoare logic with the following proof strategy:

PARALLEL ASSIGNMENT

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

COMPOSITION

$$\frac{\{p\}\ S_1\ \{r\} \quad \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$$

CONDITIONAL

$$\frac{\{p \wedge B\}\ S_1\ \{q\} \quad \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$$

WHILE

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

CONSEQUENCE

$$\frac{p \to p_1 \quad \{p_1\}\ S\ \{q_1\} \quad q_1 \to q}{\{p\}\ S\ \{q\}}$$

(basic Hoare logic)

+

SUBSTITUTION

$$\frac{\{p\}\ S\ \{q\}}{\{p[\bar{x} := \bar{y}]\}\ S\ \{q[\bar{x} := \bar{y}]\}}$$

where $\{\bar{x}\} \cap var(D \mid S) = \emptyset$ and $\{\bar{y}\} \cap change(D \mid S) = \emptyset$.

INVARIANCE

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

where $free(r) \cap change(D \mid S) = \emptyset$.

$\exists$-INTRODUCTION

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

where $\{\bar{x}\} \cap (var(D \mid S) \cup free(q)) = \emptyset$.

BLOCK

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

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

PROCEDURE CALL

$$\frac{\{p\}\ P(\bar{u})\ \{q\}}{\{p[\bar{u} := \bar{t}]\}\ P(\bar{t})\ \{q\}}$$

where $\{\bar{u}\} \cap free(q) = \emptyset$. (CBV^-)

+

RECURSION

$$\frac{\begin{array}{l}\{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash_D \{p\}\ S\ \{q\}, \\ \{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash_D \{p_1\}\ S_1\ \{q_1\}, \\ \vdots \\ \{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash_D \{p_n\}\ S_n\ \{q_n\}\end{array}}{\{p\}\ D \mid S\ \{q\}}$$

where $D = \{P_i(\bar{u}_i) :: S_i \mid i \in \{1, \ldots, n\}\}$ and $\{\bar{u}_i\} \cap free(q_i) = \emptyset$ for $i \in \{1, \ldots, n\}$.
(CBV)

Fig. 2. The proof systems: Basic Hoare logic, *CBV*$^-$, *CBV* (each includes all rules and axioms of the former).

- prove the correctness formula $\{p\}\ S\ \{q\}$ of the main statement with respect to *assumptions* $\{p'\}\ P(\bar{t})\ \{q'\}$ about the procedure calls (thus abstracting from their implementation),
- *discharge* these assumptions by proving from them correctness formulas referring to the interpretation of these procedure calls in terms of the corresponding block statements.

The above idea is formalized by the following recursion rule (also used in the textbook [4]):

$$\frac{\begin{array}{l}\{p_1\}\ P_1(\bar{t}_1)\ \{q_1\},\dots,\{p_k\}\ P_k(\bar{t}_k)\ \{q_k\} \vdash \{p\}\ S\ \{q\},\\ \{p_1\}\ P_1(\bar{t}_1)\ \{q_1\},\dots,\{p_k\}\ P_k(\bar{t}_k)\ \{q_k\} \vdash \{p_1\}\ \mathbf{begin\ local}\ \bar{u}_1 := \bar{t}_1;\ S_1\ \mathbf{end}\ \{q_1\},\\ \vdots\\ \{p_1\}\ P_1(\bar{t}_1)\ \{q_1\},\dots,\{p_k\}\ P_k(\bar{t}_k)\ \{q_k\} \vdash \{p_k\}\ \mathbf{begin\ local}\ \bar{u}_k := \bar{t}_k;\ S_k\ \mathbf{end}\ \{q_k\}\end{array}}{\{p\}\ (D \mid S)\ \{q\}},$$

where $D = \{P_i(\bar{u}_i) :: S_i \mid i \in \{1,\dots,k\}\}$. The $\vdash$ sign refers to the provability in the basic Hoare logic extended with the following rule for the block statement from the textbook [4, p. 158]:

BLOCK

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

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

Note that in the above recursion rule is allowed here that P_i and P_j denote the same procedure identifier, for some $i \neq j$. In this rule there are $k + 1$ subsidiary proofs in the premises, where k is the total number of procedure calls that appear in $(D \mid S)$. Note that the statements used on the right-hand sides of the last k provability signs $\vdash$ are the corresponding effects of inlining applied to the procedure calls on the left-hand side of $\vdash$. In this proof rule each procedure calls requires a separate subsidiary correctness proof. This results in inefficient correctness proofs.

More precisely, assuming a program $(D \mid S)$ with k procedure calls, each of the $k + 1$ subsidiary proofs in the premises of the above recursion rule can be established in the number of steps linear in the length of $(D \mid S)$. But k is linear in the length of $(D \mid S)$, as well, and as a result the bound on the length of the whole proof is quadratic in the length of $(D \mid S)$. This bound remains quadratic even for programs with a single procedure, since k remains then linear in the length of $(D \mid S)$.

Can we do better? Yes we can, by proceeding through a couple of simple steps. First, we replace each procedure call $P(\bar{t})$ such that $\bar{t} \neq \bar{u}$, where $\bar{u}$ are the formal parameters of P, by the block statement **begin local** $\bar{u} := \bar{t} : P(\bar{u})$ **end**. This gives rise to so-called *pure programs* that only contain generic procedure calls (as introduced in Section 2). For pure programs the last k premises of the above recursion rule reduce to

$$\begin{array}{l}\{p_1\}\ P_1(\bar{u}_1)\ \{q_1\},\dots,\{p_k\}\ P_k(\bar{u}_k)\ \{q_k\} \vdash\\ \qquad \{p_i\}\ \mathbf{begin\ local}\ \bar{u}_i := \bar{u}_i;\ S_i\ \mathbf{end}\ \{q_i\},\ i \in \{1,\dots,k\}.\end{array}$$

The transformation of programs the into pure ones can be avoided as follows. Incorporating in the proof system the BLOCK rule the above k premises can be reduced to

$$\begin{array}{l}\{p_1\}\ P_1(\bar{u}_1)\ \{q_1\},\dots,\{p_k\}\ P_k(\bar{u}_k)\ \{q_k\} \vdash\\ \qquad \{p_i\}\ S_i\ \{q_i\},\ i \in \{1,\dots,k\},\end{array}$$

provided that $\{\bar{u}_i\} \cap \mathit{free}(q_i) = \emptyset$ for $i \in \{1,\dots,k\}$.

Further, the replacement of every non-generic procedure call in the considered program by its corresponding block statement that uses a generic call can be captured proof theoretically by means of the proof rule

$$\frac{\{p\}\ \textbf{begin local}\ \bar{u} := \bar{t};\ P(\bar{u})\ \textbf{end}\ \{q\}}{\{p\}\ P(\bar{t})\ \{q\}}.$$

This rule can be further simplified to a direct instantiation of the generic call using a BLOCK rule:

PROCEDURE CALL

$$\frac{\{p\}\ P(\bar{u})\ \{q\}}{\{p[\bar{u} := \bar{t}]\}\ P(\bar{t})\ \{q\}},$$

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

Finally, recall that k is the number of different procedure calls that appear in $(D \mid S)$. But for pure programs $k \leq n$, where n is the number of procedure declarations in D (or $k = n$ if each procedure declared is also called).

However, we shall also need the so-called adaptation rules. The SUBSTITUTION rule from the textbook [4, p. 98] is used to deal with the occurrences in the assertions of local variables of block statements and the formal parameters of the procedure calls. Additionally, we have the rules INVARIANCE and ∃-INTRODUCTION also used in Reference [21] (though the side conditions are slightly different).

Let $\vdash_D$ refer to provability in the proof system that extends the basic Hoare logic with the above BLOCK, PROCEDURE CALL and adaptation rules. Provability depends on D to extract global information about the program variables (see the side conditions in Figure 2). This notion of provability is used in the following recursion rule:

RECURSION

$$\frac{\begin{array}{l}\{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \dots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash_D \{p\}\ S\ \{q\}, \\ \{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \dots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash_D \{p_1\}\ S_1\ \{q_1\}, \\ \vdots \\ \{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \dots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash_D \{p_n\}\ S_n\ \{q_n\}\end{array}}{\{p\}\ D \mid S\ \{q\}}$$

where $D = \{P_i(\bar{u}_i) :: S_i \mid i \in \{1, \dots, n\}\}$ and $\{\bar{u}_i\} \cap \mathit{free}(q_i) = \emptyset$ for $i \in \{1, \dots, n\}$.

We denote the resulting proof system by *CBV* (for call-by-value), and $\vdash \{p\}\ D \mid S\ \{q\}$ for provability in *CBV*, and CBV^- for the proof system *CBV* consisting of all the axioms and rules except the RECURSION rule. So $\Phi \vdash_D \{p\}\ S\ \{q\}$ refers to provability in CBV^- (under the assumptions in Φ).

Example 4.2. Consider the block statement **begin local** $u := t;\ x := u$ **end** and suppose that $x \notin \mathit{var}(t)$. We want to prove that

$$\{\textbf{true}\}\ \textbf{begin local}\ u := t;\ x := u\ \textbf{end}\ \{x = t\}. \tag{2}$$

The approach where we first prove the same pre- and postcondition for the inner statement

$$\{\textbf{true}\}\ u := t;\ x := u\ \{x = t\}$$

is not possible, since then we cannot apply the BLOCK rule, because u can occur in t. So we introduce a fresh variable u_0 and proceed as follows. Let $t' = t[u := u_0]$. By the ASSIGNMENT axiom, we have

$$\{t = t'\}\ u := t\ \{u = t'\}$$

and

$$\{u = t'\}\ x := u\ \{x = t'\},$$

so by the COMPOSITION rule, we have

$$\{t = t'\}\ u := t;\ x := u\ \{x = t'\}.$$

Now we can apply the BLOCK rule, since $u \notin \mathit{free}(x = t')$, which yields

$$\{t = t'\}\ \textbf{begin local}\ u := t;\ x := u\ \textbf{end}\ \{x = t'\}.$$

Applying the SUBSTITUTION rule on the above with the substitution $[u_0 := u]$, we then get (2) by the CONSEQUENCE rule, since $\textbf{true} \rightarrow (t = t')[u_0 := u]$ and $(x = t')[u_0 := u] \rightarrow x = t$ are valid. The crux here is that u is not changed, because it is a local variable, so we may substitute u_0 by u.

We can now also prove the correctness formula

$$\{\textbf{true}\}\ \textbf{begin local}\ u := u;\ x := u\ \textbf{end};\ \textbf{begin local}\ u := u;\ y := u\ \textbf{end}\ \{x = y\} \tag{3}$$

(as discussed in the related work section). By the argument of above, we obtain

$$\{\textbf{true}\}\ \textbf{begin local}\ u := u;\ x := u\ \textbf{end}\ \{x = u\}$$

and

$$\{\textbf{true}\}\ \textbf{begin local}\ u := u;\ y := u\ \textbf{end}\ \{y = u\}.$$

Applying to the last correctness formula the INVARIANCE rule, we get

$$\{\textbf{true} \wedge x = u\}\ \textbf{begin local}\ u := u;\ y := u\ \textbf{end}\ \{y = u \wedge x = u\}.$$

Now Equation (3) follows by the COMPOSITION and CONSEQUENCE rules. □

Example 4.3. To illustrate the use of the PROCEDURE CALL rule consider the following example. Let the set of declarations D include only the procedure declaration

$$\mathit{add}(u) :: \mathit{sum} := \mathit{sum} + u$$

and consider the correctness formula

$$\{\mathit{sum} = z\}\ D \mid \mathit{add}(\mathit{sum})\ \{\mathit{sum} = z + z\}. \tag{4}$$

So the variable *sum* is here both a global variable that is changed in the procedure body and an actual parameter of the procedure call, a feature that is not allowed in existing Hoare logics for recursive procedures, e.g., Reference [21], as explained in the introduction.

To prove Equation (4), we introduce the assumption

$$\{\mathit{sum} = z \wedge u = v\}\ \mathit{add}(u)\ \{\mathit{sum} = z + v\}$$

and first show that

$$\{\mathit{sum} = z \wedge u = v\}\ \mathit{add}(u)\ \{\mathit{sum} = z + v\} \vdash_D \{\mathit{sum} = z\}\ \mathit{add}(\mathit{sum})\ \{\mathit{sum} = z + z\}. \tag{5}$$

Applying the PROCEDURE CALL rule to the assumption, we get

$$\{\mathit{sum} = z \wedge \mathit{sum} = v\}\ \mathit{add}(\mathit{sum})\ \{\mathit{sum} = z + v\}.$$

Next, applying the SUBSTITUTION rule with the substitution $[v := z]$ we obtain

$$\{\mathit{sum} = z \wedge \mathit{sum} = z\}\ \mathit{add}(\mathit{sum})\ \{\mathit{sum} = z + z\},$$

A trivial application of the CONSEQUENCE rule then completes the proof of Equation (5).

We next show that

$$\{\mathit{sum} = z \wedge u = v\}\ \mathit{add}(u)\ \{\mathit{sum} = z + v\} \vdash_D \{\mathit{sum} = z \wedge u = v\}\ \mathit{sum} := \mathit{sum} + u\ \{\mathit{sum} = z + v\}. \tag{6}$$

First, we get by the ASSIGNMENT axiom

$$\{\mathit{sum} + u = z + v\}\ \mathit{sum} := \mathit{sum} + u\ \{\mathit{sum} = z + v\},$$

so by the CONSEQUENCE rule we get

$$\{sum = z \wedge u = v\}\ sum := sum + u\ \{sum = z + v\}.$$

From Equation (5) and Equation (6), we derive by the RECURSION rule the correctness formula (4). □

The above example proofs illustrate some characteristic uses of the adaptation rules. Adaptation rules are always applicable, and thus may lead to an arbitrary and unbounded number of applications within a proof. In Section 5, we show that we still can normalize proofs so that for each proof there exists a linear one.

5 PROOF NORMALIZATION

A proof

$$\{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash_D \{p\}\ S\ \{q\},$$

may consist of an *arbitrary* number of applications of the adaptation rules. However, we shall show in this section that we can normalize such a proof to obtain a proof that is still linear in the size of the statement S, defined as in Definition 4.1 but with the additional clauses:

- $l(P(\bar{t})) := 1$,
- $l(\textbf{begin local}\ \bar{x} := \bar{t};\ S\ \textbf{end}) := l(\bar{x} := \bar{t};\ S) + 1$.

We divide the rules of CBV^- into analytical rules and adaptation rules. Analytical rules are those rules that one applies by analyzing the structure of a specified statement: The premise or premises of analytical rules are correctness formulas of sub-statements of the statement of the conclusion. Thus the number of analytical rules one may apply in any correctness proof of a statement is bounded by the size of the statement. Adaptation can be applied on any statement, and have exactly one correctness formula as premise. So, given a derivation in the proof system CBV^-, we have that between every two successive applications of analytical rules there is a (possibly empty) sequence of adaptation rule applications.

Thus to show that one can always construct a proof that is of linear size, it suffices to transform sequences (of arbitrary length) of applications of the adaptation rules to equivalent ones of a fixed size. This transformation process (formalized below by a rewrite system) is called proof normalization, and the resulting proof is normal if all its adaptation rules are brought into a certain equivalent final form.

Rewrite system. For a formal description of the process of normalizing correctness proofs we introduce the following alphabet to abbreviate sequences of adaptation rules:

- C denotes an application of the CONSEQUENCE rule,
- S denotes an application of the SUBSTITUTION rule,
- I denotes an application of the INVARIANCE rule, and
- E denotes an application of the ∃-INTRODUCTION rule.

Sequences of adaptation rule applications are represented by the regular expression $(C \mid S \mid I \mid E)^*$, where $\mid$ denotes choice, and * is Kleene's star with respect to concatenation. Two such sequences are *equivalent* if their application to any correctness formula yields the same correctness formula. We will introduce a *rewrite system* by means of which *any* sequence in $(C \mid S \mid I \mid E)^*$ can be reduced to a equivalent *normal form* represented by the regular expression

$$[I]\ [S]\ [([C]\ E\ [C]) \mid C], \tag{7}$$

where the juxtaposition of expressions is concatenation, and $[e]$ is the usual notation of optional acceptance, i.e., $[e] = (e \mid \epsilon)$ for any regular expression e and the empty string ϵ. The regular expression (7) describes the empty sequence or non-empty sequences that consist of at most one application of the INVARIANCE followed by at most one application of the SUBSTITUTION rule that in turn is followed by a single application of the CONSEQUENCE rule or a single application of the ∃-INTRODUCTION rule possibly preceded or followed by a single application of the CONSEQUENCE rule. Examples of such expressions are *ISCEC*, *ISE*, *ISCE*, *IC*. For technical convenience, to avoid trivial applications of the CONSEQUENCE rule, we assume below some basic logical equivalences in the specification of the correctness formulas.

First, we observe that any consecutive applications of the *same* rule can be collapsed into a single such rule application. This observation is formalized by the rewrite rules $RR \longrightarrow R$, for any $R \in \{C, E, I, S\}$.

For the CONSEQUENCE rule, we have that two consecutive applications

$$\dfrac{p \to p' \qquad \dfrac{p' \to p'' \quad \{p''\}\ S\ \{q''\} \quad q'' \to q'}{\{p'\}\ S\ \{q'\}} \qquad q' \to q}{\{p\}\ S\ \{q\}}$$

can be collapsed into

$$\dfrac{p \to p'' \quad \{p''\}\ S\ \{q''\} \quad q'' \to q}{\{p\}\ S\ \{q\}} .$$

For the SUBSTITUTION rule, we have that two consecutive applications

$$\dfrac{\dfrac{\{p\}\ S\ \{q\}}{\{p[\bar{x} := \bar{y}]\}\ S\ \{q[\bar{x} := \bar{y}]\}}}{\{p[\bar{x} := \bar{y}][\bar{z} := \bar{w}]\}\ S\ \{q[\bar{x} := \bar{y}][\bar{z} := \bar{w}]\}}$$

can be collapsed into

$$\dfrac{\{p\}\ S\ \{q\}}{\{p\theta\}\ S\ \{q\theta\}},$$

where θ denotes the substitution with domain consisting of the variables of $\bar{x}$ and those variables of $\bar{z}$ that are not among $\bar{x}$ such that

- $\theta(x_i) = y_i$ if y_i does not appear in $\bar{z}$,
- $\theta(x_i) = w_j$ if y_i and z_j denote the same variable,
- $\theta(z_i) = w_i$ if z_i does not appear in $\bar{x}$.

The side-conditions are easily checked to still hold.

For the ∃-INTRODUCTION rule, we have that two consecutive applications

$$\dfrac{\dfrac{\{p\}\ S\ \{q\}}{\{\exists \bar{y} : p\}\ S\ \{q\}}}{\{\exists \bar{x} : \exists \bar{y} : p\}\ S\ \{q\}}$$

can be collapsed into

$$\dfrac{\{p\}\ S\ \{q\}}{\{\exists \bar{x}, \bar{y} : p\}\ S\ \{q\}}$$

where the side-conditions are easily checked to still hold. (Note that $\exists \bar{x}, \bar{y} : p$ and $\exists \bar{x} : \exists \bar{y} : p$ are identical.)

Finally, assuming associativity of conjunction, for the INVARIANCE rule we have that two consecutive applications

$$\frac{\dfrac{\{p\}\ S\ \{q\}}{\{p \wedge r\}\ S\ \{q \wedge r\}}}{\{(p \wedge r) \wedge r'\}\ S\ \{(q \wedge r) \wedge r'\}}$$

can be collapsed into

$$\frac{\{p\}\ S\ \{q\}}{\{(p \wedge (r \wedge r')\}\ S\ \{q \wedge (r \wedge r')\}.}$$

where the side-conditions are easily checked to still hold.

The following rewrite rules allow to move the INVARIANCE rule applications to the front.

- *CI* $\longrightarrow$ *IC*: A proof pattern

$$\frac{\dfrac{p \rightarrow p' \quad \{p'\}\ S\ \{q'\} \quad q' \rightarrow q}{\{p\}\ S\ \{q\}}}{\{p \wedge r\}\ S\ \{q \wedge r\}}$$

  consisting of an application of the CONSEQUENCE rule followed by one of the INVARIANCE rule, can be transformed into a reverse pattern

$$\frac{p \wedge r \rightarrow p' \wedge r \quad \dfrac{\{p'\}\ S\ \{q'\}}{\{p' \wedge r\}\ S\ \{q' \wedge r\}} \quad q' \wedge r \rightarrow q \wedge r}{\{p \wedge r\}\ S\ \{q \wedge r\}}.$$

- *SI* $\longrightarrow$ *IS*: A proof pattern

$$\frac{\dfrac{\{p\}\ S\ \{q\}}{\{p[\bar{x} := \bar{y}]\}\ S\ \{q[\bar{x} := \bar{y}]\}}}{\{p[\bar{x} := \bar{y}] \wedge r\}\ S\ \{q[\bar{x} := \bar{y}] \wedge r\}}$$

  consisting of an application the SUBSTITUTION rule followed by one of the INVARIANCE rule, can be transformed into a reverse pattern

$$\frac{\dfrac{\{p\}\ S\ \{q\}}{\{p \wedge r[\bar{x} := \bar{z}]\}\ S\ \{q \wedge r[\bar{x} := \bar{z}]\}}}{\{p[\bar{x} := \bar{y}] \wedge r\}\ S\ \{q[\bar{x} := \bar{y}] \wedge r\},}$$

  where $\bar{z}$ is a sequence of fresh variables of the same length as $\bar{x}$. In the SUBSTITUTION rule, we apply the (simultaneous) substitution $[\bar{x}, \bar{z} := \bar{y}, \bar{x}]$. Note that $p[\bar{x}, \bar{z} := \bar{y}, \bar{x}]$ equals $p[\bar{x} := \bar{y}]$ ($\bar{z}$ do not appear free in p) and $r[\bar{x} := \bar{z}][\bar{x}, \bar{z} := \bar{y}, \bar{x}]$ equals r.
- *EI* $\longrightarrow$ *ISE*: By the rewrite rule *ES* $\longrightarrow$ *SE* (shown below) it suffices to show that a proof pattern

$$\frac{\dfrac{\{p\}\ S\ \{q\}}{\{\exists \bar{x} : p\}\ S\ \{q\}}}{\{(\exists \bar{x} : p) \wedge r\}\ S\ \{q \wedge r\},}$$

consisting of an application of the ∃-INTRODUCTION rule followed by one application of the INVARIANCE rule, can be transformed into a pattern *IES*:

$$\cfrac{\cfrac{\cfrac{\{p\}\ S\ \{q\}}{\{p \wedge r[\bar{x} := \bar{z}]\}\ S\ \{q \wedge r[\bar{x} := \bar{z}]\}}}{\{(\exists \bar{x} : p) \wedge r[\bar{x} := \bar{z}]\}\ S\ \{q \wedge r[\bar{x} := \bar{z}]\}}}{\{(\exists \bar{x} : p) \wedge r\}\ S\ \{q \wedge r\}},$$

where, as above, $\bar{z}$ is a sequence of fresh variables of the same length as $\bar{x}$, and in the application of the SUBSTITUTION rule we apply the substitution $[\bar{z} := \bar{x}]$. Note that in the application of the ∃-INTRODUCTION rule we assume the logical equivalence between $(\exists \bar{x} : p) \wedge r[\bar{x} := \bar{z}]$ and $\exists \bar{x} : (p \wedge r[\bar{x} := \bar{z}])$, which holds, because $\bar{x}$ does not appear in $r[\bar{x} := \bar{z}]$.

The following rules allow to reverse an application of the CONSEQUENCE rule, or an application of the ∃-INTRODUCTION rule, d by an application of the SUBSTITUTION rule.

- *CS* ⟶ *SC*: A proof pattern

$$\cfrac{\cfrac{p \to p' \quad \{p'\}\ S\ \{q'\} \quad q' \to q}{\{p\}\ S\ \{q\}}}{\{p[\bar{x} := \bar{y}]\}\ S\ \{q[\bar{x} := \bar{y}]\}}.$$

consisting of an application of the CONSEQUENCE rule followed by one of the SUBSTITUTION rule, can be transformed into a reverse pattern

$$\cfrac{p[\bar{x} := \bar{y}] \to p'[\bar{x} := \bar{y}] \quad \cfrac{\{p'\}\ S\ \{q'\}}{\{p'[\bar{x} := \bar{y}]\}\ S\ \{q'[\bar{x} := \bar{y}]\}} \quad q'[\bar{x} := \bar{y}] \to q[\bar{x} := \bar{y}]}{\{p[\bar{x} := \bar{y}]\}\ S\ \{q[\bar{x} := \bar{y}]\}}.$$

- *ES* ⟶ *SE*: A proof pattern

$$\cfrac{\cfrac{\{p\}\ S\ \{q\}}{\{\exists \bar{z} : p\}\ S\ \{q\}}}{\{(\exists \bar{z} : p)[\bar{x} := \bar{y}]\}\ S\ \{q[\bar{x} := \bar{y}]\}},$$

consisting of an application of the ∃-INTRODUCTION rule followed by one of the SUBSTITUTION rule, can be transformed in the reverse pattern

$$\cfrac{\cfrac{\{p\}\ S\ \{q\}}{\{p[\bar{x} := \bar{y}]\}\ S\ \{q[\bar{x} := \bar{y}]\}}}{\{\exists \bar{z} : (p[\bar{x} := \bar{y}])\}\ S\ \{q[\bar{x} := \bar{y}]\}},$$

where we note that we may assume that $\bar{z}$ and $\bar{x}$ are disjoint (note that we can simply remove the overlapping variables from the substitution $[\bar{x} := \bar{y}]$) and $\bar{z}$ and $\bar{y}$ are disjoint (by an alphabetic conversion of the existentially quantified variables $\bar{z}$), and thus $(\exists \bar{z} : p)[\bar{x} := \bar{y}]$ is logically equivalent to $\exists \bar{z} : (p[\bar{x} := \bar{y}])$.

The mutual dependencies between consecutive applications of the CONSEQUENCE and the ∃-INTRODUCTION rules do not allow a simple reversal of the ordering as above. This is because, on the one hand, the CONSEQUENCE rule may enable an application of the ∃-INTRODUCTION rule by generating a postcondition that does not contain free occurrences of the quantified variable. However, the ∃-INTRODUCTION may enable an application of the CONSEQUENCE rule by *weakening* the precondition. However, these dependencies can be unravelled by the following rewrite rule.

- *ECE* ⟶ *CEC*: A proof pattern

$$\frac{\dfrac{p \rightarrow \exists \bar{x} : p' \qquad \dfrac{\{p'\}\ S\ \{q'\}}{\{\exists \bar{x} : p'\}\ S\ \{q'\}} \qquad q' \rightarrow q}{\{p\}\ S\ \{q\}}}{\{\exists \bar{y} : p\}\ S\ \{q\}}$$

can be transformed into a pattern (assuming without loss of generality that $\bar{x}$ and $\bar{y}$ are disjoint)

$$\frac{\exists \bar{y} : p \rightarrow \exists \bar{x}, \bar{y} : p' \qquad \dfrac{\dfrac{p' \rightarrow p' \qquad \{p'\}\ S\ \{q'\} \qquad q' \rightarrow q}{\{p'\}\ S\ \{q\}}}{\{\exists \bar{x}, \bar{y} : p'\}\ S\ \{q\}} \qquad q \rightarrow q}{\{\exists \bar{y} : p\}\ S\ \{q\}}.$$

Note that indeed we can existentially quantify both the variables $\bar{x}$ and $\bar{y}$ in this single application of the SUBSTITUTION rule, because the above pattern *ECE* ensures that the variables of $\bar{x}$ do not appear free in q' and that the variables $\bar{y}$ do not appear free in q. But validity of $q' \rightarrow q$ implies that we may assume without loss of generality that q also does not contain free occurrences of $\bar{x}$. Further, note that validity of $p \rightarrow \exists \bar{x} : p'$ implies that of $\exists \bar{y} : p \rightarrow \exists \bar{x}, \bar{y} : p'$.

We thus obtain a rewrite system (see Reference [32]), in which the objects are sequences of adaptation rule applications, and rewrite rules are applicable anywhere in such sequences. It is straightforward to check that no rewrite rule is applicable if and only if the sequence is in normal form, as defined by the above regular expression (7). To show that any sequence in $(C \mid S \mid I \mid E)^*$ can be reduced to a equivalent normal form it thus suffices to show that the rewrite system *terminates.*

Termination. We show that our rewrite system terminates (see Reference [32, Chapter 6]) by associating with each sequence in $(C \mid S \mid I \mid E)^*$ an element of a *well-founded ordering* that (strictly) decreases with each application of a rewrite rule. Relevant parameters to consider are

- the number of occurrences of C, S, I and E,
- for each occurrence of I the number of *preceding* occurrences of C, S and E,
- for each occurrence of S the number of *preceding* occurrences of C and E.

However, the rules *EI* ⟶ *IES* and *ECE* ⟶ *CEC* increases the number of occurrences of S and C, and as such these rules will in general lead to an increase of the above second and third parameters. Therefore, we single out the two parameters that record for each occurrence of I the number of preceding occurrences of E and the number of occurrences of E itself. To ensure each application of a rewrite rule causes an overall decrease, we use pairs of natural numbers, ordered *lexicographically*, that is, $\langle x_1, x_2 \rangle \leq \langle y_1, y_2 \rangle$ if $x_1 < y_1$, or $x_1 = y_1$ and $x_2 \leq y_2$, and associate with each sequence in $(C \mid S \mid I \mid E)^*$ a pair $\langle x, y \rangle$ such that x sums up

- the number of occurrences of E,
- for each occurrence of I the number of preceding occurrences of E,

and y sums up

- the number of occurrences of C, S an I,
- for each occurrence of I the number of preceding occurrences of C and S,
- for each occurrence of S the number of preceding occurrences of C and E.

It is straightforward to check that with each application of a rewrite rule the above measure indeed decreases. Consider as a typical example the rule $CI \longrightarrow IC$. Clearly the number of occurrences of C preceding the occurrence of I to which this rewrite rule is applied decreases. It further does not affect the other parameters. So it decreases the above y component, which causes an overall decrease, because the above x component is unaffected. As another example consider the rule $EI \longrightarrow IES$. Since this rule generates a new S, it may *increase* the above y component. However, the number of occurrences of E preceding the occurrence of I to which this rewrite rule is applied decreases, and as such it decreases the above x component (which thus causes an overall decrease). A similar argument applies to the rule $ECE \longrightarrow CEC$: The increase of the number of occurrences of C is compensated by the decrease of the number of occurrences of E.

We also have checked termination of our rewriting system using the AProVE tool [20]. However, the automatic termination proof is not as legible as the description given above.

Next, we show that the rewrite system is confluent, which implies that for any sequence in $(C \mid S \mid I \mid E)^*$ there exists a *unique* normal form.

Confluence. By Newman's Lemma, every rewrite system that is terminating and weakly confluent is also confluent (see Reference [32]). Thus, to show that our rewrite system is confluent, it suffices to show that it is weakly confluent. The Critical Pair Lemma states that a rewrite system is weakly confluent if and only if all its critical pairs are *convergent* (also called joinable, see Reference [32, Section 2.7.2]): The two reducible expressions of the critical pair have a common reduct. We thus establish confluence by analyzing all critical pairs that arise from our rewrite rules. This road to confluence is also known as the Knuth-Bendix Theorem.

We now show that all critical pairs are convergent. These critical pairs are identified by considering all ways in which the left-hand side patterns (the redexes) of the rewrite rules can overlap. Let P, Q, and R range over $\{C, S, I, E\}$, and (overlapping) redexes) be indicated by underlining/overlining. For all pairs PQ and QR of redexes of two letters, we obtain the critical pair $\underline{P}\underline{\overline{Q}}\overline{R}$: It thus suffices to analyze for each redex of two letters all critical pairs obtained by either adding one letter after or adding one before. For the single redex ECE consisting of three letters, it suffices to analyze $\underline{EC}\underline{\overline{E}}\overline{CE}$, which is the single way it can overlap with itself, and analyze the remaining critical pairs $\underline{E}\underline{\overline{E}}\overline{CE}$ and $\underline{EC}\underline{\overline{E}}\overline{P}$ for P in $\{S, I, E\}$. This analysis is covered by the following cases (in some cases below the transitive closure $\longrightarrow^*$ abstracts from the order of applications of rewrite rules that do not have overlapping redexes).

- The critical pair $\underline{P}\underline{\overline{P}}\overline{P}$ is convergent, for $P \in \{C, S, I, E\}$:
  Applying the rewrite rule $PP \longrightarrow P$ to the first redex or the second one, both give PP.
- The critical pairs $\underline{P}\underline{\overline{P}}\overline{Q}$ and $\underline{P}\underline{\overline{Q}}\overline{Q}$ are convergent, where $P \in \{C, S\}$ and $Q = I$, or $P \in \{C, E\}$ and $Q = S$:
  - Applying first the rewrite rule $PP \longrightarrow P$ to the first redex of $\underline{P}\underline{\overline{P}}\overline{Q}$, gives
    $\underline{PP}Q \longrightarrow \underline{PQ} \longrightarrow QP$.
  - Applying first the rewrite rule $PQ \longrightarrow QP$ to the second redex of $\underline{P}\underline{\overline{P}}\overline{Q}$, gives
    $P\underline{PQ} \longrightarrow \underline{PQ}P \longrightarrow Q\underline{PP} \longrightarrow QP$.
  - Applying first the rewrite rule $PQ \longrightarrow QP$ to the first redex of $\underline{P}\underline{\overline{Q}}\overline{Q}$, gives
    $\underline{PQ}Q \longrightarrow Q\underline{PQ} \longrightarrow \underline{QQ}P \longrightarrow QP$.

  - Applying first the rewrite rule $QQ \longrightarrow Q$ to the second redex of $\underline{P\overline{Q}}\overline{Q}$, gives $P\underline{QQ} \longrightarrow \underline{PQ} \longrightarrow QP$.
- The critical pair $\underline{E\overline{I}}\overline{I}$ is convergent:
  - Applying first the rewrite rule $EI \longrightarrow ISE$ to the first redex, gives $\underline{EI}I \longrightarrow IS\underline{EI} \longrightarrow I\underline{SI}SE \longrightarrow^{*} ISE$.
  - Applying first the rewrite rule $II \longrightarrow I$ to the second redex, gives $E\underline{II} \longrightarrow \underline{EI} \longrightarrow ISE$.
- The critical pair $\underline{E\overline{E}}\overline{I}$ is convergent:
  - Applying first the rewrite rule $EE \longrightarrow E$ to the first redex, gives $\underline{EE}I \longrightarrow \underline{EI} \longrightarrow ISE$.
  - Applying first the rewrite rule $EI \longrightarrow ISE$ to the second redex, gives $E\underline{EI} \longrightarrow \underline{EI}SE \longrightarrow IS\underline{ES}E \longrightarrow^{*} ISE$.
- The critical pair $\underline{E\overline{S}}\overline{I}$ is convergent:
  - Applying first the rewrite rule $ES \longrightarrow SE$ to the first redex, gives $\underline{ES}I \longrightarrow S\underline{EI} \longrightarrow \underline{SI}SE \longrightarrow^{*} ISE$.
  - Applying first the rewrite rule $SI \longrightarrow IS$ to the second redex, gives $E\underline{SI} \longrightarrow \underline{EI}S \longrightarrow IS\underline{ES} \longrightarrow^{*} ISE$.
- The critical pair $\underline{EC\overline{E}}\overline{E}$ is convergent:
  - Applying first the rewrite rule $ECE \longrightarrow CEC$ to the first redex, gives $\underline{ECE}E \longrightarrow C\underline{ECE} \longrightarrow \underline{CC}EC \longrightarrow CEC$.
  - Applying first the rewrite rule $EE \longrightarrow E$ the second redex, gives $EC\underline{EE} \longrightarrow \underline{ECE} \longrightarrow CEC$.
- The critical pair $\underline{E\overline{E}}\overline{CE}$ is convergent:
  - Applying first the rewrite rule $EE \longrightarrow E$ the first redex, gives $\underline{EE}CE \longrightarrow \underline{ECE} \longrightarrow CEC$.
  - Applying first the rewrite rule $ECE \longrightarrow CEC$ to the second redex, gives $E\underline{ECE} \longrightarrow \underline{ECE}C \longrightarrow CE\underline{CC} \longrightarrow CEC$.
- The critical pair $\underline{EC\overline{E}}\overline{S}$ is convergent:
  - Applying first the rewrite rule $ECE \longrightarrow CEC$ to the first redex, gives $\underline{ECE}S \longrightarrow CE\underline{CS} \longrightarrow^{*} SCEC$.
  - Applying first the rewrite rule $ES \longrightarrow SE$ to the second redex, gives $EC\underline{ES} \longrightarrow E\underline{CS}E \longrightarrow^{*} S\underline{ECE} \longrightarrow SCEC$.
- The critical pair $\underline{EC\overline{E}}\overline{I}$ is convergent:
  - Applying first the rewrite rule $ECE \longrightarrow CEC$ to the first redex, gives $\underline{ECE}I \longrightarrow CE\underline{CI} \longrightarrow C\underline{EI}C \longrightarrow \underline{CI}SEC \longrightarrow^{*} ISCEC$.
  - Applying first the rewrite rule $EI \longrightarrow ISE$ to the second redex, gives $EC\underline{EI} \longrightarrow E\underline{CI}SE \longrightarrow \underline{EI}CSE \longrightarrow^{*} IS\underline{ECE} \longrightarrow ISCEC$.
- The critical pair $\underline{EC\overline{E}}\overline{CE}$ is convergent:
  - Applying first the rewrite rule $ECE \longrightarrow CEC$ to the first redex, gives $\underline{ECE}CE \longrightarrow CE\underline{CCE} \longrightarrow C\underline{ECE} \longrightarrow \underline{CC}EC \longrightarrow CEC$.
  - Applying first the rewrite rule $ECE \longrightarrow CEC$ to the second redex, gives $EC\underline{ECE} \longrightarrow E\underline{CC}EC \longrightarrow \underline{ECE}C \longrightarrow CE\underline{CC} \longrightarrow CEC$.

We also have checked confluence of our rewriting system using the CSI tool [27]. However, also here the automatic confluence proof is not as legible as the description given above.

Scheduling strategy. For any sequence, we can derive the normal form of Equation (7) by scheduling the rewrite rules as follows:

(1) For any sequence of letters $(C \mid S \mid I \mid E)^*$, move I to the front and collapse all consecutive I applications.
(2) For the resulting postfix $(C \mid S \mid E)^*$, move S to the front and collapse all consecutive S applications.
(3) For the resulting non-empty postfix $(C \mid E)^*$, first collapse all consecutive E applications to a single E, and all consecutive C applications to a single C. Then we repeatedly apply the rewrite rule $ECE \longrightarrow CEC$ and collapse the possibly generated consecutive C applications until at most one E application is left (with possibly one C before and possibly one C after it).

Thus starting from any sequence of adaptation rules, we end up with at most five rule applications.

Linear proofs. Using the proof normalization procedure above, we can now prove the claim that our proof system admits linear proofs. Given a set of contracts Φ, denote by

$$\Phi \vdash_D^k \{p\}\ S\ \{q\}$$

the existence of a proof of $\{p\}\ S\ \{q\}$ in CBV^- from the assumptions Φ that consists of at most k applications of axioms and proof rules.

Lemma 5.1. *Suppose that* $\Phi \vdash_D \{p\}\ S\ \{q\}$. *Then* $\Phi \vdash_D^k \{p\}\ S\ \{q\}$, *where* $k \leq 6 \times l(S)$.

Proof. The proof consists of at least $l(S)$ applications of analytical axioms and rules, and between two analytical rules, or between an analytical rule and an assumption or an axiom application, we have an arbitrary sequence of adaptations. The latter can be reduced to a sequence of at most five rule applications. Thus the total number of axiom and rule applications needed in a proof is at most $6 \times l(S)$. □

We denote by $\vdash^k \{p\}\ D \mid S\ \{q\}$ the existence of a proof of $\{p\}\ D \mid S\ \{q\}$ in CBV that consists of at most k applications of axioms and proof rules.

Theorem 5.2. *Let* $D = \{P_i(\bar{u}_i) :: S_i \mid i \in \{1, \ldots, n\}\}$ *and suppose that* $\vdash \{p\}\ D \mid S\ \{q\}$. *Then* $\vdash^{O(k)} \{p\}\ D \mid S\ \{q\}$, *where* k *is the length of* $(D \mid S)$ *seen as a string of symbols.*

Proof. The correctness formula $\{p\}\ D \mid S\ \{q\}$ was established using the RECURSION rule. By Lemma 5.1 to prove the premises of this rule, it takes at most $6 \times (l(S) + \Sigma_{i=1}^n l(S_i))$ steps, where $S_1, \ldots, S_n$ are the bodies of the procedures declared in D. Clearly, $l(S) + \Sigma_{i=1}^n l(S_i)$ is less than the length of $(D \mid S)$ seen as a string of symbols. □

In the above result, we measure the complexity in size of the proof, being the number of axiom and rule applications. By inspection of the rewrite rules introduced above, it is easily seen that the weight of the proof (being the maximum complexity of the pre- and postconditions that occur within correctness formulas as they appear in rule applications, as measured by the number of connectives and quantifiers) remains constant: thus, the existence of a linear proof does not result in an explosion of the complexity of assertions.

6 SOUNDNESS

We first prove the soundness of CBV^-. We treat soundness of the block rule and the rule for procedure calls in the context of a set of declarations (soundness of the rules for the remaining statements are standard).

Lemma 6.1 (Soundness of the Block Rule). *Suppose that*

$$\models \{p\}\ D \mid \bar{x} := \bar{t};\ S\ \{q\},$$

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

$$\models \{p\}\ D \mid \textbf{begin local}\ \bar{x} := \bar{t};\ S\ \textbf{end}\ \{q\}.$$

Proof. Follows directly from Corollary 3.6(i). □

Lemma 6.2 (Soundness of the Procedure Call Rule). *Suppose that*

$$\models \{p\}\ D \mid P(\bar{u})\ \{q\},$$

where $P(\bar{u}) :: S \in D$ *and* $\{\bar{u}\} \cap \mathit{free}(q) = \emptyset$*. Then*

$$\models \{p[\bar{u} := \bar{t}]\}\ D \mid P(\bar{t})\ \{q\}$$

Proof. By Lemma 3.2 and Corollary 3.6(ii)

$$\models \{p\}\ D \mid P(\bar{u})\ \{q\} \text{ iff} \models \{p\}\ D \mid \bar{u} := \bar{u};\ S\ \{q\} \text{ iff} \models \{p\}\ D \mid S\ \{q\}$$

and

$$\models \{p[\bar{u} := \bar{t}]\}\ D \mid P(\bar{t})\ \{q\} \text{ iff} \models \{p[\bar{u} := \bar{t}]\}\ D \mid \bar{u} := \bar{t};\ S\ \{q\}.$$

But by the soundness of PARALLEL ASSIGNMENT axiom $\models\ \{p[\bar{u} := \bar{t}]\}\ D \mid \bar{u} := \bar{t}\ \{p\}$, so $\models \{p\}\ D \mid S\ \{q\}$ implies by the soundness of the COMPOSITION rule $\models \{p[\bar{u} := \bar{t}]\}\ D \mid \bar{u} := \bar{t};\ S\ \{q\}$. □

We next introduce a semantic interpretation of $\Phi \vdash_D \{p\}\ S\ \{q\}$ that abstracts from the actual implementation of the procedures as specified by D. Note that in such a proof only information about the variables of D is used (in the INVARIANCE, SUBSTITUTION and ∃-INTRODUCTION rules). We therefore introduce the notation $D' \leq D$ to denote that $\mathit{var}(D' \mid \textbf{skip}) \subseteq \mathit{var}(D \mid \textbf{skip})$ and for every $P(\bar{u}) :: S \in D$ we have $\mathit{change}(D' \mid P(\bar{u})) \subseteq \mathit{change}(D \mid P(\bar{u}))$. Note that $D' \leq D$ implies that $\mathit{change}(D' \mid S) \subseteq \mathit{change}(D \mid S)$, for every statement S. It also holds that $D \leq D$, and $D^k \leq D$ for every kth approximation of D (see Definition 3.1). We use the relation $\leq$ between programs in the soundness proof below, since syntactic approximations do not add more variables or variables that may be changed.

Further, let $\models_D \Phi$ denote that $\models \{p\}\ D \mid P(\bar{u})\ \{q\}$, for every $\{p\}\ P(\bar{u})\ \{q\} \in \Phi$.

Definition 6.3. $\Phi \models_D \{p\}\ S\ \{q\}$ denotes that $\models_{D'} \Phi$ implies $\models \{p\}\ D' \mid S\ \{q\}$, for all $D' \leq D$.

We then have the following corollary.

Corollary 6.4 (Soundness of *CBV*$^-$). $\Phi \vdash_D \{p\}\ S\ \{q\}$ *implies* $\Phi \models_D \{p\}\ S\ \{q\}$.

Proof. The proof proceeds by a straightforward induction on the length of proof, using the above lemmas for the BLOCK rule and the PROCEDURE CALL rule. The side conditions of the SUBSTITUTION and INVARIANCE rule are dealt with by the assumption that $\mathit{change}(D' \mid P(\bar{u})) \subseteq \mathit{change}(D \mid P(\bar{u}))$, for every $P(\bar{u}) \in D$. □

To prove soundness of the proof system *CBV* it then suffices to prove the soundness of the RECURSION rule (note that the RECURSION rule is the only rule for deriving a correctness formula about a program).

Theorem 6.5 (Soundness of the Recursion Rule). *Let* $D = \{P_1(\bar{u}_1) :: S_1, \ldots, P_n(\bar{u}_n) :: S_n\}$ *and* $\Phi = \{\{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \ldots, \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\}\}$ *be such that* $\{\bar{u}_i\} \cap \mathit{free}(q_i) = \emptyset$*,* $i \in \{1, \ldots, n\}$*. Then*

$$\Phi \models_D \{p\}\ S\ \{q\} \text{ and } \Phi \models_D \{p_i\}\ S_i\ \{q_i\},\ i \in \{1, \ldots, n\}$$

implies

$$\models \{p\}\ D \mid S\ \{q\}.$$

Proof. By Definition 6.3, the conclusion follows from the first premise and that we have $D \leq D$, if we show $\models_D \Phi$. For the latter, it suffices to show that $\models \{p_i\}\ D \mid P_i(\bar{u}_i)\ \{q_i\}$, for $i \in \{1, \ldots, n\}$. By the **Approximation** item of Lemma 3.2 this reduces to showing that $\models \{p_i\}\ D^k \mid P_i(\bar{u}_i)\ \{q_i\}$, for every $k \geq 0$ and $i \in \{1, \ldots, n\}$. We prove this by induction on k.

Base case: By Definition 3.1 and the **Inlining** item of Lemma 3.2 it suffices to observe that $\mathcal{M}[\![D^0 \mid P_i(\bar{u}_i)]\!] = \mathcal{M}[\![D^0 \mid \Omega]\!]$ and that $\models \{p'\}\ D^0 \mid \Omega\ \{q'\}$, for any assertions p' and q'.

Induction step: We show that the induction hypothesis $\models_{D^k} \Phi$ implies $\models_{D^{k+1}} \Phi$. Since we have the second premise $\Phi \models_D \{p_i\}\ S_i\ \{q_i\}$ and $D^k \leq D$, the induction hypothesis implies $\models \{p_i\}\ D^k \mid S_i\ \{q_i\}$ by Definition 6.3. It is sufficient to show that the latter is equivalent to

$$\models \{p_i\}\ D^{k+1} \mid P_i(\bar{u}_i)\ \{q_i\}.$$

By Definition 3.1, Corollary 3.6(*ii*), and $\{\bar{u}_i\} \cap \mathit{free}(q_i) = \emptyset$, we have that

$$\models \{p_i\}\ D^{k+1} \mid P_i(\bar{u}_i)\ \{q_i\}$$

if and only if

$$\models \{p_i\}\ D^{k+1} \mid \bar{u}_i := \bar{u}_i; S_i^{k+1}\ \{q_i\}$$

for every $i \in \{1, \ldots, n\}$. Observe that

$$\mathcal{M}[\![D^{k+1} \mid \bar{u}_i := \bar{u}_i; S_i^{k+1}]\!] = \mathcal{M}[\![D^{k+1} \mid S_i^{k+1}]\!] = \mathcal{M}[\![D^{k+1} \mid S_i[S_1^k/P_1, \ldots, S_n^k/P_n]]\!]$$

holds by simplification and definition of S_i^{k+1}. Further, we have that

$$\mathcal{M}[\![D^{k+1} \mid S_i[S_1^k/P_1, \ldots, S_n^k/P_n]]\!] = \mathcal{M}[\![D^k \mid S_i[S_1^k/P_1, \ldots, S_n^k/P_n]]\!] = \mathcal{M}[\![D^k \mid S_i]\!],$$

since $S_i[S_1^k/P_1, \ldots, S_n^k/P_n]$ does not contain any procedure calls (thus allowing us to replace all declarations), and by the definition of D^k and the **Inlining** item of Lemma 3.2. Thus we have established that

$$\models \{p_i\}\ D^{k+1} \mid P_i(\bar{u}_i)\ \{q_i\}$$

if and only if

$$\models \{p_i\}\ D^k \mid S_i\ \{q_i\}.$$

□

7 COMPLETENESS

We now prove completeness of *CBV* in the sense of Cook, that is, assuming that the language $\mathcal{L}$ is expressive (see Definition 3.8) and that all valid assertions can be used in proofs. To this end, following Gorelick [21], we introduce the *most general correctness formulas.* Given a set of declarations D, in our setup these are contracts of the form

$$\{\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}\}\ P(\bar{u})\ \{\exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S)\},$$

where

- $P(\bar{u}) :: S \in D$,
- $\{\bar{x}\} = \mathit{change}(D \mid P(\bar{u}))$,
- $\bar{v}$ and $\bar{z}$ are lists of fresh variables, of the same length as, respectively, $\bar{u}$ and $\bar{x}$,
- $SP(p, D \mid S)$ is the strongest postcondition given in Definition 3.7.

The crucial difference between our most general correctness formulas and the ones used in [21] is that in our case the strongest postcondition of the generic call $P(\bar{u})$ is defined in terms of a strongest postcondition of the body S *in which the formal parameters are quantified out.* This has to do with the way we model the call-by-value parameter mechanism by a general notion of local variables, whereas, as already mentioned, in Reference [21] the call-by-name parameter mechanism is used in combination with a specific notion of local variables that ensures static scoping semantically by variable renaming.

To compensate for this loss of your information, we have introduced the variables $\bar{v}$ that in the above postcondition denote the initial values of the formal parameters $\bar{u}$. The variables $\bar{z}$ are used in the above postcondition to denote the initial values of the variables $\bar{x}$ that could be changed by a call $(D \mid P(\bar{u}))$. In particular, note that $\bar{u}$ and $\bar{x}$ have no variables in common, because $\{\bar{x}\} = change(D \mid P(\bar{u})) = change(D \mid \textbf{begin local } \bar{u} := \bar{u};\ S \textbf{ end}) = change(D \mid S) \setminus \{\bar{u}\}$, where $P(\bar{u}) :: S \in D$. In fact, it is important to note that the sets of variables listed in $\bar{u}, \bar{v}, \bar{x}$ and $\bar{z}$ are mutually disjoint.

The following crucial theorem shows that the assertion

$$(\bar{t} = \bar{v})[\bar{x} := \bar{z}] \wedge \exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S)$$

encodes the semantics of a call $(D \mid P(\bar{t}))$, where P is declared by $P(\bar{u}) :: S \in D$. Note that since the sets of variables $\bar{v}$ and $\bar{x}$ are disjoint, $(\bar{t} = \bar{v})[\bar{x} := \bar{z}]$ equals $\bar{t}[\bar{x} := \bar{z}] = \bar{v}$. Further, since $\bar{z}$ are used to denote in the postcondition the initial values of the variables $\bar{x}$, $\bar{t}[\bar{x} := \bar{z}]$ denotes in the postccondition the initial values of the actual parameters $\bar{t}$.

Theorem 7.1. *Let P be declared by $P(\bar{u}) :: S \in D$. Further, let the lists of variables $\bar{v}, \bar{x}$ and $\bar{z}$ be defined as above and let*

$$\sigma \models (\bar{t} = \bar{v})[\bar{x} := \bar{z}] \wedge \exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S). \tag{8}$$

It follows that

$$\mathcal{M}[\![D \mid P(\bar{t})]\!](\sigma'') = \{\sigma\},$$

where $\sigma'' = \sigma[\bar{x} := \sigma(\bar{z})]$.

Proof. From the assumption (8) it follows that

$$\sigma' \models SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S),$$

where $\sigma' = \sigma[\bar{u} := \bar{d}]$ for some sequence $\bar{d}$ of elements (of the domain of I).

By the definition of the strongest postcondition there exists a state σ_0 such that

- $\sigma_0 \models \bar{x} = \bar{z} \wedge \bar{u} = \bar{v}$ and
- $\mathcal{M}[\![D \mid S]\!](\sigma_0) = \{\sigma'\}$.

Let $\sigma'' = \sigma[\bar{x} := \sigma(\bar{z})]$. Assume first that

$$\sigma''[\bar{u} := \sigma''(\bar{t})] = \sigma_0. \tag{9}$$

Then, given that $\sigma''(\bar{u}) = \sigma(\bar{u})$ and $\sigma'[\bar{u} := \sigma(\bar{u})] = \sigma$, Corollary 3.3 allows us to derive the following chain of equalities:

$$\begin{aligned} &\mathcal{M}[\![D \mid P(\bar{t})]\!](\sigma'') \\ =\ &\Big(\mathcal{M}[\![D \mid S]\!](\sigma''[\bar{u} := \sigma''(\bar{t})])\Big)[\bar{u} := \sigma''(\bar{u})] \\ =\ &\Big(\mathcal{M}[\![D \mid S]\!](\sigma_0)\Big)[\bar{u} := \sigma''(\bar{u})] \\ =\ &\Big(\mathcal{M}[\![D \mid S]\!](\sigma_0)\Big)[\bar{u} := \sigma(\bar{u})] \\ =\ &\{\sigma'[\bar{u} := \sigma(\bar{u})]\} = \{\sigma\}. \end{aligned}$$

Figure 3 clarifies the relation between the introduced states and illustrates the construction of a computation of $P(\bar{t})$ that starts in the state σ'' with the final state σ and that corresponds to the above equalities.

It remains to prove Equation (9). First, note the following consequences of the **Access and Change** item of Lemma 3.2 that follow from the fact that the variables listed in $\bar{v}$ and $\bar{z}$ are fresh and that by the choice of $\bar{x}$ we have $change(D) \subseteq \{\bar{u}\} \cup \{\bar{x}\}$:

$$\begin{tikzcd}
\sigma_0 \arrow[r, "S"] & \sigma' \arrow[d, shift left, "{[\bar{u} := \sigma''(\bar{u})]}"] \\
\sigma'' \arrow[u, "{[\bar{u} := \sigma''(\bar{t})]}"] & \sigma \arrow[l, "{[\bar{x} := \sigma(\bar{z})]}"] \arrow[u, shift left, "{[\bar{u} := \bar{d}]}"]
\end{tikzcd}$$

Fig. 3. Construction of a computation of $P(\bar{t})$.

(a) $\sigma'(\bar{v}) = \sigma_0(\bar{v})$,
(b) $\sigma'(\bar{z}) = \sigma_0(\bar{z})$,
(c) $\sigma'(y) = \sigma_0(y)$, where $y \notin \{\bar{u}\} \cup \{\bar{x}\}$.

By Equation (8), $\sigma \models (\bar{t} = \bar{v})[\bar{x} := \bar{z}]$, so by the definition of σ'' (this is an instance of the Simultaneous Substitution Lemma in the textbook [4, p. 50]) $\sigma'' \models \bar{t} = \bar{v}$, i.e.,

$$\sigma''(\bar{t}) = \sigma''(\bar{v}).$$

This, item (a) above, the fact that the sets of variables listed in $\bar{u}$, $\bar{v}$ and $\bar{x}$ are mutually disjoint, and the definitions of σ'' and σ' justifies the following chain of equalities:

$$\sigma''[\bar{u} := \sigma''(\bar{t})](\bar{u}) = \sigma''(\bar{t}) = \sigma''(\bar{v}) = \sigma(\bar{v}) = \sigma'(\bar{v}) = \sigma_0(\bar{v}) = \sigma_0(\bar{u}).$$

Next, the same observations and item (b) above justifies the following chain of equalities:

$$\sigma''[\bar{u} := \sigma''(\bar{t})](\bar{x}) = \sigma''(\bar{x}) = \sigma(\bar{z}) = \sigma'(\bar{z}) = \sigma_0(\bar{z}) = \sigma_0(\bar{x}).$$

Finally, take a variable $y \notin \{\bar{u}\} \cup \{\bar{x}\}$. Then the above observations and item (c) above justifies the following chain of equalities:

$$\sigma''[\bar{u} := \sigma''(\bar{t})](y) = \sigma''(y) = \sigma(y) = \sigma'(y) = \sigma_0(y).$$

We thus established Equation (9), which concludes the proof. □

From the above theorem, we derive the following corollary.

Corollary 7.2. *Let $\models \{p\}\ D \mid P(\bar{t})\ \{q\}$, where P is declared by $P(\bar{u}) :: S \in D$. Further, let the lists of variables $\bar{v}$, $\bar{x}$ and $\bar{z}$ be defined as above and let*

$$Inv \equiv (p \wedge \bar{t} = \bar{v})[\bar{x} := \bar{z}].$$

It follows that $\models (Inv \wedge \exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S)) \rightarrow q$.

Proof. Take a state σ such that

$$\sigma \models Inv \wedge \exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S).$$

By the above Theorem 7.1 it follows that

$$\mathcal{M}[\![D \mid P(\bar{t})]\!](\sigma'') = \{\sigma\},$$

where $\sigma'' = \sigma[\bar{x} := \sigma(\bar{z})]$. We are given that $\sigma \models p[\bar{x} := \bar{z}]$, so $\sigma'' \models p$ by the definition of σ''. (This is again the contents of the Simultaneous Substitution Lemma in the textbook [4, p. 50].) We conclude by the assumption $\models \{p\}\ D \mid P(\bar{t})\ \{q\}$ that $\sigma \models q$. □

In the completeness proof below, we transform a given precondition p of a procedure call $D \mid P(\bar{t})$ into an *invariant* $(p \wedge \bar{t} = \bar{v})[\bar{x} := \bar{z}]$. The following corollary shows that any precondition p logically implies its invariant transformation for some values of the variables $\bar{v}$ and $\bar{z}$.

Corollary 7.3. *Let the lists of variables $\bar{v}, \bar{x}$ and $\bar{z}$ be defined as above and let*

$$Inv \equiv (p \wedge \bar{t} = \bar{v})[\bar{x} := \bar{z}].$$

Suppose that the variables listed in $\bar{v}$ and $\bar{z}$ do not appear in p. Then

$$\models p \rightarrow \exists \bar{v}, \bar{z} : (Inv \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v}).$$

Proof. Take a state σ such that $\sigma \models p$. By assumption the variables listed in $\bar{v}$ and $\bar{z}$ do not occur in p, so $\tau \models p$, where $\tau = \sigma[\bar{v}, \bar{z} := \sigma(\bar{t}), \sigma(\bar{x})]$.

Note that $\tau[\bar{x} := \tau(\bar{z})] = \tau$, so $\tau[\bar{x} := \tau(\bar{z})] \models p$. By the Simultaneous Substitution Lemma of Reference [4, p. 50] $\tau \models p[\bar{x} := \bar{z}]$, i.e.,

$$\sigma[\bar{v}, \bar{z} := \sigma(\bar{t}), \sigma(\bar{x})] \models p[\bar{x} := \bar{z}].$$

Further, for an arbitrary σ we have

$$\sigma[\bar{v}, \bar{z} := \sigma(\bar{t}), \sigma(\bar{x})] \models (\bar{t} = \bar{v})[\bar{x} := \bar{z}] \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v},$$

so

$$\sigma[\bar{v}, \bar{z} := \sigma(\bar{t}), \sigma(\bar{x})] \models (p \wedge \bar{t} = \bar{v})[\bar{x} := \bar{z}] \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v}.$$

This shows that

$$\models p \rightarrow \exists \bar{v}, \bar{z} : ((p \wedge \bar{t} = \bar{v})[\bar{x} := \bar{z}] \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v})$$

and yields the conclusion by the definition of *Inv*. □

As in Reference [21] the completeness proof is based on the following key lemma (see also Reference [3, pp. 450–452]). We denote by $G(D)$ the set of most general correctness formulas for all procedures declared in D.

Lemma 7.4. *Suppose that $\models \{p\}\ D \mid T\ \{q\}$. Then $G(D) \vdash_D \{p\}\ T\ \{q\}$.*

Proof. As in Reference [21] and Reference [5], we proceed by induction on the structure of the statement T, with two essential cases being different.

Block statements. Suppose that $\models \{p\}\ D \mid \textbf{begin local}\ \bar{y} := \bar{t};\ S\ \textbf{end}\ \{q\}$. Let $\bar{y}'$ be some fresh variables corresponding to the local variables $\bar{y}$. Note that by the definition of *change*$(D \mid S)$,

$$\{\bar{y}\} \cap change(D \mid \textbf{begin local}\ \bar{y} := \bar{t};\ S\ \textbf{end}) = \emptyset. \tag{10}$$

So from the soundness of the INVARIANCE rule it follows that

$$\models \{p \wedge \bar{y} = \bar{y}'\}\ D \mid \textbf{begin local}\ \bar{y} := \bar{t};\ S\ \textbf{end}\ \{q \wedge \bar{y} = \bar{y}'\}.$$

Consequently, by the soundness of the CONSEQUENCE rule

$$\models \{p \wedge \bar{y} = \bar{y}'\}\ D \mid \textbf{begin local}\ \bar{y} := \bar{t};\ S\ \textbf{end}\ \{q[\bar{y} := \bar{y}']\},$$

because $q \wedge \bar{y} = \bar{y}'$ implies $q[\bar{y} := \bar{y}']$. From Corollary 3.6(i) it then follows that

$$\models \{p \wedge \bar{y} = \bar{y}'\}\ D \mid \bar{y} := \bar{t};\ S\ \{q[\bar{y} := \bar{y}']\}.$$

By the induction hypothesis we obtain

$$G(D) \vdash_D \{p \wedge \bar{y} = \bar{y}'\}\ \bar{y} := \bar{t};\ S\ \{q[\bar{y} := \bar{y}']\}.$$

An application of the BLOCK rule then gives

$$G(D) \vdash_D \{p \wedge \bar{y} = \bar{y}'\} \textbf{ begin local } \bar{y} := \bar{t};\ S \textbf{ end } \{q[\bar{y} := \bar{y}']\}.$$

Thanks to Equation (10), we can now apply the SUBSTITUTION rule with the substitution $[\bar{y}' := \bar{y}]$ and subsequently the CONSEQUENCE rule to replace $p \wedge \bar{y} = \bar{y}$ by p. This yields

$$G(D) \vdash_D \{p\} \textbf{ begin local } \bar{y} := \bar{t};\ S \textbf{ end } \{q\}.$$

Procedure calls. Suppose that $\models \{p\}\ D \mid P(\bar{t})\ \{q\}$, where P is declared by $P(\bar{u}) :: S \in D$.

Assume the most general correctness formula

$$\{\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}\}\ P(\bar{u})\ \{\exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S)\}.$$

By applying the SUBSTITUTION rule to rename the variables listed in $\bar{v}$ and $\bar{z}$, we may assume these variables not appear in $\{p\}\ P(\bar{t})\ \{q\}$. By the choice of the list $\bar{x}$, $\bar{z}$ and $\bar{v}$ none of them contains a variable from $\bar{u}$. So by the PROCEDURE CALL rule with the substitution $[\bar{u} := \bar{t}]$, we obtain

$$\{\bar{x} = \bar{z} \wedge \bar{t} = \bar{v}\}\ P(\bar{t})\ \{\exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S)\}.$$

As in Corollary 7.3 let

$$Inv \equiv (p \wedge \bar{t} = \bar{v})[\bar{x} := \bar{z}].$$

Note that $free(Inv) \cap \{\bar{x}\} = \emptyset$ and by definition $change(D \mid P(\bar{t})) = change(D \mid P(\bar{u})) = \{\bar{x}\}$, so $free(Inv) \cap change(D \mid P(\bar{t})) = \emptyset$. Thus by the INVARIANCE rule

$$\{Inv \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v}\}P(\bar{t})\{Inv \wedge \exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S)\}. \tag{11}$$

It remains to show that the pre- and postconditions of the above correctness formula can be replaced, respectively, by p and q.

First, we have by Corollary 7.2

$$\models (Inv \wedge \exists \bar{u} : SP(\bar{x} = \bar{z} \wedge \bar{u} = \bar{v}, D \mid S)) \rightarrow q,$$

so by the CONSEQUENCE rule we obtain from Equation (11)

$$\{Inv \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v}\}\ P(\bar{t})\ \{q\}.$$

By assumption the variables in $\bar{v}$ and $\bar{z}$ do not appear in $P(\bar{t})$ or q, so by the $\exists$-INTRODUCTION rule

$$\{\exists \bar{v}, \bar{z} : (Inv \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v})\}\ P(\bar{t})\ \{q\}.$$

By Corollary 7.3

$$\models p \rightarrow \exists \bar{v}, \bar{z} : (Inv \wedge \bar{x} = \bar{z} \wedge \bar{t} = \bar{v}),$$

so we obtain the desired conclusion by the CONSEQUENCE rule.

The remaining cases are as in Reference [11]. □

THEOREM 7.5 (COMPLETENESS). *The proof system CBV is complete (in the sense of Cook).*

PROOF. Let $\models \{p\}\ D \mid S\ \{q\}$. To prove $\{p\}\ D \mid S\ \{q\}$ in *CBV*, we use the RECURSION rule with $G(D)$ as the set of assumptions in the subsidiary proofs. Lemma 7.4 ensures the first premise. The remaining n premises also follow by this lemma. Indeed, suppose

$$G(D) = \{\{\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i\}\ P_i(\bar{u}_i)\ \{\exists \bar{u}_i : SP(\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i, D \mid S_i)\} \mid i \in \{1, \ldots, n\}\},$$

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

Choose an arbitrary $i \in \{1, \ldots, n\}$. By the definition of the strongest postcondition

$$\models \{\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i\}\ D \mid S_i\ \{SP(\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i, D \mid S_i)\},$$

so

$$\models \{\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i\}\ D \mid S_i\ \{\exists \bar{u} : SP(\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i, D \mid S_i)\}$$

by the soundness of the CONSEQUENCE rule. Hence by Lemma 7.4

$$G(D) \vdash_D \{\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i\}\ S_i\ \{\exists \bar{u} : SP(\bar{x}_i = \bar{z}_i \wedge \bar{u}_i = \bar{v}_i, D \mid S_i)\}\}.$$

We now obtain $\vdash \{p\}\ D \mid S\ \{q\}$ by the RECURSION rule. □

We conclude this section with the following simple calculation of the complexity of the completeness proof (measured in terms of the number of axioms and rules applied). Since the above completeness proof shows that the INVARIANCE, SUBSTITUTION and ∃-INTRODUCTION rules are only used for the block statements and the procedure calls, we can lower the upper bound of Theorem 5.2 by

$$2 \times x_1 + 3 \times x_2 + 6 \times x_3,$$

where x_1 denotes the number of assignments, block statements and iteration statements, x_2 denotes the number of block statements, and x_3 denotes the number of procedure calls.

8 FORMALIZATION IN COQ

The main semantic argument underlying the completeness proof is the proof of Theorem 7.1 (and its Corollaries 7.2 and 7.3), which shows how the semantics of procedure calls can be expressed by the strongest postcondition. We checked this proof by means of the Coq theorem prover, and to this end we only need to formalize the semantics of the programming language and the strongest postcondition. Our formalization is publicly and freely accessible.[3]

Theorem 7.1 requires first of all a formalization of the syntax and semantics of the programming language and the assertion language. A main challenge is to abstract from syntax to avoid the overhead of irrelevant details, while still capturing essential syntactical notions like occurrences of variables and substitution. We formalize in Coq expressions semantically as particular functions from states to (abstract) values. Similarly, we formalize assertions semantically, as sets of states. Moreover, this extensional approach[4] to the formalization of expressions and assertions also allows to abstract from the given interpretation I and the assumption of the expressibility of the strongest postcondition. However, and this is where our approach crucially differs from, for example, References [34, 36], to capture essential syntactic notions like that of occurrences of free variables (this is required in the notion of freshness used in Theorem 7.1.) we introduce *finitely-based* functions and predicates as the semantic structures for expressions and assertions, as explained in more detail next.

An expression is formalized as a function e from states to values together with an additional list of variables $\bar{x}$ with the *coincidence condition* that for every σ, τ such that $\sigma[\bar{x}] = \tau[\bar{x}]$ we have $e(\sigma) = e(\tau)$. Similarly, an assertion consists of a set of states X and a list of variables $\bar{x}$ with the coincidence condition that for every σ, τ such that $\sigma[\bar{x}] = \tau[\bar{x}]$ we have $\sigma \in X$ iff $\tau \in X$. For assertion p, we write $\sigma \models p$ to mean $\sigma \in X$. The intuition is that the finite set of variables associated with an expression or assertion forms its basis: the semantic counterpart of the syntactical notion of free variable occurrences. Only the values assigned by a state to variables in its basis can influence an expression's value or an assertion's truth. Assertions are equal whenever their extension and basis are equal, regardless of the proof of the coincidence condition.[5]

[3] See Reference [22]: https://doi.org/10.5281/zenodo.4005507

[4] This is also known as a shallow embedding.

[5] For this, we may use proof irrelevance, which in Coq follows from propositional extensionality.

For the formalization, we further need the following constructions of expressions and assertions. For a given value, we immediately can construct an expression that evaluates to that value with the empty basis. For a given variable, we can construct an expression that evaluates to the value of that variable and its basis consists of (at least) the given variable. Since one could always extend a basis, we take the smallest. Given two expressions, we can construct the assertion that states their equality: Its extension consists of those states in which the equality is evident, and its basis is the union of the bases of the equated expressions. We also have equality of lists of variables of equal length. For all these constructions, we verified the coincidence condition.

Further, we have the constructions of conjunction, implication, substitution, and existential quantification assertions. Given two assertions p and q, the conjunction $p \wedge q$ consists of the intersection of the states of p and q and as basis the union of the bases of p and q. Similarly, we can define the implication $p \rightarrow q$. For the substitution construct $p[\bar{x} := \bar{t}]$, we take an assertion p and assignment $\bar{x} := \bar{t}$ that is an association of variables in $\bar{x}$ to expressions in $\bar{t}$. Let $\sigma[\bar{x} := \bar{t}]$ be a state update replacing the values of variables $\bar{x}$ by the corresponding value of $\bar{t}$. We define the set of states of the substitution $p[\bar{x} := \bar{t}]$ as $\{\sigma \mid \sigma[\bar{x} := \sigma(\bar{t})] \models p\}$. The basis of the substitution construct is the basis of p with variables in $\bar{x}$ removed, together with the union of the bases of expressions in $\bar{t}$. Thus, if a variable occurs both in $\bar{x}$ and in some expression in $\bar{t}$, then it is still in the basis of $p[\bar{x} := \bar{t}]$. The Simultaneous Substitution Lemma, i.e., $\sigma \models p[\bar{x} := \bar{t}]$ if and only if $\sigma[\bar{x} := \sigma(\bar{t})] \models p$, then can be verified already by definition. Finally, the existential quantification $\exists \bar{x} : p$ is formalized by the set of states $\{\sigma \mid \exists \tau : \sigma[\bar{x} := \tau(\bar{x})] \models p\}$ and as basis, the basis of p with all variables in $\bar{x}$ removed.

The syntax of statements we formalize does not include **while** statements, but does include a **diverge** statement that is needed for Approximation part of Lemma 3.1. This modification does not reduce expressivity of the language, as it is well known that while loops can be encoded using (tail) recursion, but it simplifies the definition of, and proofs about, the semantics of programs.

We have formalized the big-step operational semantics as given in Figure 1 by an inductive predicate, so that every program $(D \mid S)$ is associated to a relation of states denoted by $(\_, D \mid S) \Rightarrow \_$. This relation can also be treated as a function mapping states to a subset of states. We then have formally verified the properties of Lemma 3.1, Corollary 3.2, Lemma 3.3, and Corollary 3.4. Not all of these results are needed in Theorem 7.1, but their verification increases our confidence that the formalized semantics contains no error.

Given an assertion p and program $(D \mid S)$, the strongest postcondition $SP(p, D \mid S)$ consists of an extension *and* a basis for which the coincidence condition holds. Its extension is defined by $\{\sigma' \mid \exists \sigma : (\sigma, D \mid S) \Rightarrow \sigma' \text{ and } \sigma \models p\}$, and as its basis $\bar{x}$, we take the union of the set of variables occurring in the program and the basis of p. The set of variables occurring in a program can be defined inductively by the structure of the program; it is the union of the bases of all expressions and all variables that occur in the program. Now, to show that the coincidence condition holds for the strongest postcondition, we need to show that for every state σ' in its extension, any state τ' obtained from σ' in which variables not in the basis are possibly modified are also in the extension:

$$\sigma' \models SP(p, D \mid S) \text{ and } \sigma'[\bar{x}] = \tau'[\bar{x}] \text{ implies } \tau' \models SP(p, D \mid S).$$

To show this property, it is sufficient to show a more general closure property on the meaning of programs. The following lemma establishes the latter property.

Lemma 8.1. *Given a list of variables $\bar{x}$ such that every variable occurring in $(D \mid S)$ is also in $\bar{x}$, and given that $\sigma'[\bar{x}] = \tau'[\bar{x}]$ and $(\sigma, D \mid S) \Rightarrow \sigma'$, then $(\tau'[\bar{x} := \sigma(\bar{x})], D \mid S) \Rightarrow \tau'$.*

By $\tau'[\bar{x} := \sigma(\bar{x})]$, we mean a state, which has the value in σ for the variables in $\bar{x}$, and the value in τ' otherwise. Intuitively, we can reconstruct a run backwards from a final state that simulates the original run of the program but with different values stored in non-essential variables.

The above formalization allows for a formal verification of the statements of Theorem 7.1, without adding any custom axioms to the Coq theorem prover. We have used the axioms of functional extensionality, propositional extensionality, and classical logic, all provided by the base system. Functional extensionality is necessary to show equality of states, which are total functions from variables to values. Functional and propositional extensionality is needed for showing (and using) the fact that our semantics is compositional. Classical logic is typically used when reasoning in Hoare logic. This concludes the description of the formalization effort of Reference [22].

9 CONCLUSIONS

We conclude that the proof system presented in this article and its relative completeness result consolidates and improves the proof complexity of the proof system in the textbook [4] and Reference [5]. Following this, it would be interesting to revisit related work.

The already-mentioned work of Clarke [10] led to a research on proof systems for programming languages in which in presence of static scope both nested declarations of mutually recursive procedures and procedure names as parameters of procedure calls were allowed. In particular in Reference [30], Reference [13], and Reference [19], sound and relatively complete Hoare-like proof systems were established, each time under different assumptions concerning the assertion language. In all these papers the call-by-name parameter mechanism was used and the actual parameters were variables. It would be interesting to see how to modify these proofs to programming languages in which the call-by-value parameter mechanism is used instead of the call-by-name mechanism.

It would be useful to extend (in an appropriate sense) the results of this article to total correctness of programs. In this context, we should mention [2], where a sound and relatively complete (in an appropriate sense) proof system for total correctness of programs in presence of recursive procedures was provided. However, in the article only procedures without parameters were considered.

Formally verified completeness and soundness theorems of Hoare logic in theorem provers have been established in, for example, Reference [28]. In future work, we intend to proceed in further formalizing the (syntactic) proof system as described in this article and verifying its soundness and relative completeness theorems in Coq. Such work is comparable to Von Oheimb's thesis [35], who formalized a Hoare logic for reasoning about a subset of the Java language in Isabelle/HOL.

We can extend our completeness result to object-oriented Java-like programs by the syntax-directed transformation from object-oriented programs to recursive programs as described in [5]. However, separation logic (see Reference [29]) is particularly tailored to modular reasoning about aliasing as it arises in object (and pointer) structures. Completeness for recursive *parameterless* procedures in separation logic is established in Reference [17]. Of interest is that to obtain this completeness result the *frame rules*, which are important for the local reasoning in separation logic are *not* needed (as observed by the authors themselves). A slight modification of the usual adaptation rules for reasoning about invariance are sufficient. Therefore, we expect that the completeness result of Reference [17] can be extended to recursive procedures with call-by-value parameter passing, using our techniques.

ACKNOWLEDGMENT

Krzysztof R. Apt identified the problem of completeness of reasoning about call-by-value in Hoare logic. We also thank him for his contributions to earlier versions of the article. We thank

Ernst-Rüdiger Olderog and Stijn de Gouw for helpful comments about the contributions of a number of relevant references. We further are most grateful for the comments of the anonymous referees that led to various improvements.

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification—The KeY Book: From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6

[2] P. America and F. S. de Boer. 1990. Proving total correctness of recursive procedures. *Inf. Comput.* 84, 2 (1990), 129–162.

[3] K. R. Apt. 1981. Ten years of hoare's logic, a survey, part I. 3, 4 (1981), 431–483.

[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog. 2009. *Verification of Sequential and Concurrent Programs* (3rd ed.). Springer-Verlag.

[5] Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog, and Stijn de Gouw. 2012. Verification of object-oriented programs: A transformational approach. *J. Comput. Syst. Sci.* 78, 3 (2012), 823–852.

[6] Krzysztof R. Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare's logic. *Form. Aspects Comput.* 31, 6 (2019), 751–807. https://doi.org/10.1007/s00165-019-00501-3

[7] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.* 7, 3 (2005), 212–232.

[8] Robert Cartwright and Derek C. Oppen. 1981. The logic of aliasing. *Acta Inf.* 15, 4 (1981), 365–384.

[9] Adam Chlipala. 2013. *Certified Programming with Dependent Types*. MIT Press.

[10] E. M. Clarke. 1979. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *J. ACM* 26, 1 (1979), 129–147.

[11] S. A. Cook. 1978. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.* 7, 1 (1978), 70–90.

[12] S. A. Cook. 1981. Corrigendum: Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.* 10, 3 (1981), 612.

[13] W. Damm and B. Josko. 1983. A sound and relatively complete Hoare-logic for a language with higher type procedures. *Acta Inf.* 20, 1 (1983), 59–101.

[14] J. W. de Bakker. 1979. A sound and complete proof system for partial program correctness. In *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science (Lecture Notes in Computer Science)*, Vol. 74. Springer, 1–12.

[15] J. W. de Bakker. 1980. *Mathematical Theory of Program Correctness*. Prentice-Hall International.

[16] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. 2019. Verifying OpenJDK's sort method for generic collections. *J. Autom. Reas.* 62, 1 (2019), 93–126.

[17] Mahmudul Faisal Al Ameen and Makoto Tatsuta. 2016. Completeness for recursive procedures in separation logic. *Theor. Comput. Sci.* 631 (2016), 73–96. https://doi.org/10.1016/j.tcs.2016.04.004

[18] M. Foley and C. A. R. Hoare. 1971. Proof of a recursive program: Quicksort. *Comput. J.* 14, 4 (1971), 391–395.

[19] Steven M. German, Edmund M. Clarke, and Joseph Y. Halpern. 1989. Reasoning about procedures as parameters in the language L4. *Inf. Comput.* 83, 3 (1989), 265–359.

[20] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2004. Automated termination proofs with AProVE. In *Proceedings of the International Conference on Rewriting Techniques and Applications*. Springer, 210–220.

[21] G. A. Gorelick. 1975. *A Complete Axiomatic System for Proving Assertions about Recursive and Non-recursive Programs*. Master's thesis. University of Toronto.

[22] Hans-Dieter A. Hiep. 2020. *Completeness and Complexity of Reasoning about Call-by-value in Hoare Logic (Proof Files)*. Zenodo. https://doi.org/10.5281/zenodo.4005508

[23] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580, 583.

[24] C. A. R. Hoare. 1971. Procedures and parameters: An axiomatic approach. In *Proceedings of Symposium on the Semantics of Algorithmic Languages,* Lecture Notes in Mathematics, Vol. 188. Springer-Verlag, 102–116.

[25] Peter E. Lauer. 1971. *Consistent Formal Theories of the Semantics of Programming Languages*. Ph.D. Dissertation. Queen's University Belfast, UK.

[26] Bertrand Meyer. 1992. Applying "design by contract." *IEEE Comput.* 25, 10 (1992), 40–51.

[27] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. 2017. CSI: New evidence—A progress report. In *Proceedings of the Annual Conference on Automated Deduction (CADE'26)*. Springer, 385–397.

[28] Tobias Nipkow. 2002. Hoare logics for recursive procedures and unbounded nondeterminism. In *Proceedings of the International Workshop on Computer Science Logic,* Lecture Notes in Computer Science, Vol. 2471. Springer.

[29] Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.

[30] E.-R. Olderog. 1984. Correctness of programs with pascal-like procedures without global variables. *Theor. Comput. Sci.* 30 (1984), 49–90.

[31] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2020. *Programming Language Foundations*. Electronic textbook. Version 5.8. Available at https://softwarefoundations.cis.upenn.edu.

[32] Terese, Marc Bezem, Jan Willem Klop, Roel de Vrijer, et al. 2003. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, Vol. 55. Cambridge University Press.

[33] Gauthier van den Hove. 2015. On the origin of recursive procedures. *Comput. J.* 58, 11 (2015), 2892–2899. https://doi.org/10.1093/comjnl/bxu145

[34] David von Oheimb. 1999. Hoare logic for mutual recursion and local variables. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'99)* Lecture Notes in Computer Science, Vol. 1738. Springer, 168–180. https://doi.org/10.1007/3-540-46691-6_13

[35] David von Oheimb. 2001. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. Ph.D. Dissertation. Technische Universität München. Available at http://ddvo.net/diss/.

[36] David von Oheimb and Tobias Nipkow. 2002. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In *Proceedings of the Formal Methods—Getting IT Right (FME'02)* Lecture Notes in Computer Science, Vol. 2391. Springer, 89–105. https://doi.org/10.1007/3-540-45614-7_6

Received September 2019; revised May 2021; accepted July 2021