Operational Annotations
A New Method for Sequential Program Verification

Paul C. Attie(✉)

School of Computer and Cyber Sciences, Augusta University, Augusta, Georgia
PATTIE@augusta.edu

Abstract. I present a new method for specifying and verifying the partial correctness of sequential programs. The key observation is that, in Hoare logic, assertions are used as selectors of states, that is, an assertion specifies the set of program states that satisfy the assertion. Hence, the usual meaning of the partial correctness Hoare triple $\{f\}\,P\,\{g\}$: if execution is started in *any of the states that satisfy assertion* f, then, upon termination, the resulting state will be *some state that satisfies assertion* g. There are of course other ways to specify a set of states. Given a program α, the *post-states* of α are the states that α may terminate in, given that α starts executing in an arbitrary initial state. I introduce the *operational triple* $[\alpha]\,P\,[\beta]$ to mean: if execution of P is started in *any post-state of* α, then upon termination, the resulting state will be *some post-state of* β. Here, α is the *pre-program*, and plays the role of a pre-condition, and β is the *post-program*, and plays the role of a post-condition.

Keywords: Program verification · Hoare logic

1 Introduction

I present a system for verifying partial correctness of sequential programs. In contrast to Floyd-Hoare logic [6,10], I do not use pre-conditions and post-conditions, but rather *pre-programs* and *post-programs*. An assertion is essentially a means for defining a set of states: those for which the assertion evaluates to true. Hence the usual Hoare triple $\{f\}\,P\,\{g\}$ means that if execution of P is started in *any of the states that satisfy assertion* f, then, upon termination of P, the resulting state will be *some state that satisfies assertion* g. Another method of defining a set of states is with a sequential program α which starts execution in any state, i.e., with precondition *true*. The set of states in which α terminates (taken over all possible starting states) constitues the set of states that α defines. I call these the *post-states* of α.

I introduce the *operational triple* $[\alpha]\,P\,[\beta]$, in which α is the *pre-program*, β is the *post-program*, and P is the program being verified. The meaning of $[\alpha]\,P\,[\beta]$ is as follows. Consider executions of α that start in any state. From the final state of every such execution, P is executed. Let φ be the set of resulting final

© Springer Nature Switzerland AG 2022
J. V. Deshmukh et al. (Eds.): NFM 2022, LNCS 13260, pp. 597–615, 2022.
https://doi.org/10.1007/978-3-031-06773-0_32

states of P. That is, φ results from executing P from any post-state of α. Also, let ψ be the set of post-states of β, i.e., the set of final states that result from executing β starting in any state. Then, $[\alpha]\, P\, [\beta]$ is defined to mean $\varphi \subseteq \psi$. That is, the post-states of $\alpha; P$ are a subset of the post-states of β.

Contributions of the Paper. This paper makes the following contributions:

- *Code derivation/synthesis via trading*: Starting with a pre-program α and post-program β, parts of α can be "traded" into the actual program P, since they are both written in code. This gives a method of deriving program code from specification code, e.g., unwinding an outer loop of the pre-program and then trading it into the program can give initial code for a loop body of the program. This technique is illustrated in Sect. 6. Trading gives flexibility in developing both the program and the pre-program, as code can be freely moved between the program and the pre-program, and in some cases there is benefit to moving code from the program back into the pre-program; see [1] for examples of this. The Trading tactic is not available in logic-based verification methods, e.g., Floyd-Hoare logic [6,10] and separation logic [25].
- *Separation effect*: To deal with pointer-based structures required the extension of Hoare logic to separation logic [25]. For example, in the in-place reversal of a linked list, a key requirement is that the initial list not contain cycles. This requirement is expressible in my framework by a pre-program which a priori constructs an acyclic linked list. Section 6 gives such a pre-program: it declares an array n of `Node` objects (thereby making them all distinct) and then scans the array setting $n[i]$ to point to $n[i+1]$ (thereby constructing an acyclic linked list).
- *Practical application*: Since the pre-program α and the post-program β are not actually executed, they can be written without concern for efficiency, and they can refer to any well defined expression, e.g., $\delta[t]$ for the shortest path distance from a designated source s to node t. Nevertheless, being expressed as code, they may be easier for developers to write than logic specifications, since code is a formalism that developers are already well familiar with.

2 Related Work

The use of assertions to verify programs was introduced by Floyd [6] and Hoare [10]: a precondition f expresses what can be assumed to hold before execution of a program P, and a postcondition g expresses what must hold afterwards. The "Hoare triple" $\{f\}\, P\, \{g\}$ thus states that if f holds when execution of P starts, then g will hold upon termination of P. If termination is not required, this is known as *partial correctness*, and if termination is required we have *total correctness*. Both precondition and postcondition are expressed as a formula of a suitable logic, e.g., first order logic. Subsequently, Dijkstra introduced the *weakest precondition predicate transformer* [5]: $wp(P, g)$ is the weakest predicate f whose truth before execution of P guarantees termination of P in a state satisfying g. He then used weakest preconditions to define a method for formally deriving

a program from a specification, expressed as a precondition-postcondition pair. Later, Hoare observed that the Hoare triple can be expressed operationally, when he wrote "$\{p\}\, q\, \{r\} \triangleq p; q < r$" in [11], but he does not seem to have developed this observation into a proof system.

The formalization of specifications and program correctness has lead to a rich and extensive literature on program verification and refinement. Hoare's original rules [10] were extended to deal with non-determinism, fair selection, and procedures [7]. Separation logic [25] was devised to deal with pointer-based structures.

A large body of work deals with the notion of *program refinement* [2,20]: start with an initial artifact, which serves as a specification, and gradually refine it into an executable and efficient program. This proceeds incrementally, in a sequence of refinement steps, each of which preserves a "refinement ordering" relation $\sqsubseteq$, so that we have $P_0 \sqsubseteq \ldots \sqsubseteq P_n$, where P_0 is the initial specification and P_n is the final program. Morgan [20] starts with a pre-condition/post-condition specification and refines it into an executable program using rules that are similar in spirit to Dijkstra's weakest preconditions [5]. Back and Wright [2] use *contracts*, which consist of assertions (failure to hold causes a breach of the contract), assumptions (failure to hold causes vacuous satisfaction of the contract), and executable code. As such, contracts subsume both pre-condition/post-condition pairs and executable programs, and so serve as an artifact for the seamless refinement of a pre-condition/post-condition specification into a program.

A related development has been the application of monads to programming [18,19]. A monad is an endofunctor T over a category C together with a unit natural transformation from 1_C (the identity functor over C) to T and a multiplication natural transformation from T^2 to T. The Hoare state monad contains Hoare triples (precondition, program, postcondition) [12], and a computation maps an initial state to a pair consisting of a final state and a returned value. The unit is the monadic operation return, which lifts returned values into the state monad, and the multiplication is the monadic operation bind, which composes two computations, passing the resulting state and returned value of the first computation to the second [28]. The Dijkstra monad captures functions from postconditions to preconditions [12,27]. The return operation gives the weakest precondition of a pure computation, and the bind operation gives the weakest precondition for a composition of two computations.

Hoare logic and weakest preconditions are purely assertional proof methods. Monads combine operational and assertional techniques, since they provide operations which return the assertions that are used in the correctness proofs. My approach is purely operational, since it uses no assertions (formula in a suitable logic) but rather pre- and post-programs instead. My approach thus represents the operational endpont of the assertional–operational continuum, with Hoare logic/weakest preconditions at the other (assertional) endpoint, and monads somewhere in between.

3 Syntax and Semantics of the Programming Language

I use a basic programming language consisting of primitive types, arrays, and reference types, assignments, if statements, while loops, for loops, procedure definition and invocation, class definition, object creation and referencing. I assume standard primitive types (integers, boolean etc.) and the usual semantics for reference types: object identifiers are pointers to the object, and the identity of an object is given by its location in memory, so that two objects are identical iff they occupy the same memory. Parameter passing is by value, but as usual a passed array/object reference allows the called procedure to manipulate the original array/object.

My syntax is standard and self-explanatory. I also use $[]$ to denote nondeterministic choice between two commands [9]. For integers i, j with $i \leqslant j$, I use $x := [i : j]$ as syntactic sugar for $x := i \,[]\, \cdots \,[]\, x := j$, i.e., a random assignment of a value in $i, \ldots, j$ to x. This plays the role of the range assertion $i \leqslant x \leqslant j$ in Hoare logic. I use *tt* for true, *ff* for false, and *skip* for the statement that terminates immediately with no change of state.

My proof method relies on (1) the axioms and inference rules introduced in this paper, and (2) an underlying method for establishing program equivalence. Any semantics in which the above are valid can be used. For concreteness, I assume a standard small-step (SOS) operational semantics [24,26].

An ***execution*** of program P is a finite sequence $s_0, s_1, \ldots, s_n$ of states such that (1) s_i results from a single small step of P in state s_{i-1}, for all $i \in 1, \ldots, n$, (2) s_0 is an initial state of P, and (3) s_n is a final (terminating) state of P. A ***behavior*** of program P is a pair of states (s, t) such that (1) $s_0, s_1, \ldots, s_n$ is an execution of P, $s = s_0$, and $t = s_n$. Write $\{\!|P|\!\}$ for the set of behaviors of P.

4 Operational Annotations

I use a sequential program to specify a set of states. There is no constraint on the initial states, and the specified set is the set of all possible final states. If any initialization of variables is required, this must be done explicitly by the program.

Definition 1 (Post-state set). *For a terminating program P,*

$$post(P) \triangleq \{t \mid (\exists\, s : (s, t) \in \{\!|P|\!\})\}\,.$$

That is, $post(P)$ is the set of all possible final states of P, given any initial state. Infinite (nonterminating) executions of P do not contribute to the post-state set. The complete state set is specified by the program *skip*, and the empty state set by the program `while` (tt) *skip*. The central definition of the paper is that of *operational triple* $[\alpha]\, P\, [\beta]$:

Definition 2 (Operational triple). *Let α, P, and β be programs. Then*

$$[\alpha]\, P\, [\beta] \;\triangleq\; post(\alpha; P) \subseteq post(\beta).$$

Recall the meaning of $[\alpha]\,P\,[\beta]$: every terminating execution of $\alpha; P$ ends in a state that is also a final state of some terminating execution of β. Thus $[\alpha]\,P\,[\beta]$ specifies *partial correctness*, since it deals only with terminating executions, and permits P to have non-terminating executions. Even α and β can have non-terminating executions; these do not change the meaning of the specification, since they do not change the post-state set. If α (β) has only nonterminating executions then $post(\alpha) = \emptyset$, ($post(\beta) = \emptyset$), which is akin to a false precondition (postcondition). So if α or P (or both) has only nonterminating executions, then $[\alpha]\,P\,[\beta]$ is vacuously true. If β has only nonterminating executions, then $[\alpha]\,P\,[\beta]$ is false, unless α or P (or both) has only nonterminating executions.

4.1 Program Ordering and Equivalence

The next section presents a deductive system for establishing validity of operational triples. The rules of inference use three kinds of hypotheses: (1) operational triples (over "substatements" as usual), and (2) program ordering assertions $P \preceq Q$, and (3) program equivalence assertions $P \equiv Q$.

Definition 3 (Program ordering, $\preceq$). $P \preceq Q \triangleq post(P) \subseteq post(Q)$.

Here P is "stronger" than Q since it has fewer post-states (w.r.t., the precondition tt, i.e., all possible pre-states), and so P produces an output which satisfies, in general, more constraints than the output of Q. This is *not* the same as the usual program refinement relation, since the mapping from pre-states to post-states induced by the execution of P is not considered. Also, the "direction" of the inclusion relation is reversed w.r.t. the usual refinement ordering, where we write $Q \sqsubseteq P$ to denote that "P refines Q", i.e., P satisfies more specifications than Q.

The operational triple is expressible as program ordering, since by Definition 3, $[\alpha]\,P\,[\beta] \triangleq \alpha; P \preceq \beta$.

If one increases the set of states in which a program can start execution, then the set of states in which the program terminates is also possibly increased, and is certainly not decreased. That is, the set of post-states is monotonic in the set of pre-states. Since prefixing a program α with another program γ simply restricts the states in which α starts execution, I have the following.

Proposition 1. $\gamma; \alpha \preceq \alpha$.

Definition 4 (Program equivalence, $\equiv$). *Programs P and Q are* equivalent *iff they have the same behaviors*: $P \equiv Q \triangleq \{\!|P|\!\} = \{\!|Q|\!\}$.

That is, I take as program equivalence the equality of program behaviors. Note that equivalence is *not* ordering in both directions. This discrepancy is because ordering is used for weakening/strengthening laws (and so post-state inclusion is sufficient) while equivalence is used for substitution, and so, for programs at least, equality of behaviors is needed.

Any method for establishing program ordering and equivalence is sufficient for my needs. The ordering and equivalence proofs in this paper are informal, and

based on obvious concepts such as the commutativity of assignment statements that modify different variables/objects.

Future work includes investigating proof systems for program equivalence [3,4,13,22,23]. Some of these are mechanized, and some use bisimulation and circular reasoning. I will look into using these works for formally establishing program equivalence hypotheses needed in my examples (which are then akin to Hoare logic verification conditions), and to adapting these systems to establish program ordering, e.g., replace bisimulation by simulation.

5 A Deductive System for Operational Annotations

Table 1 presents a deductive system for operational annotations. Soundness of the system is formally established in the full version of the paper [1]. I do not provide a rule for the ***for*** loop, since it can be easily turned into a ***while***. The following are informal intuition for the axioms and inference rules.

Sequence Axiom. If P executes after pre-program α, the result is identical to post-program $\alpha; P$, i.e., the sequential composition of α and P. This gives an easy way to calculate a post-program for given pre-program and program. The corresponding Hoare logic notion, namely the strongest postcondition, is easy to compute (in closed form) only for straight-line code.

Empty Pre-program. Program P is "doing all the work", and so the resulting post-program is also P. Having *skip* as a pre-program is similar to having tt as a precondition in Hoare logic.

Empty Program. This is analogous to the axiom for *skip* in Hoare logic: $\{f\}$ *skip* $\{f\}$, since *skip* has no effect on the program state.

Trading Rule. Sequential composition is associative: $\alpha; (P1; P2) \equiv (\alpha; P1); P2$. By Definitions 1, 4: $post(\alpha; (P1; P2)) = post((\alpha; P1); P2)$. Hence, if the program is a sequential composition $P1; P2$, I can take $P1$ and add it to the end of the pre-program α. I can also go in the reverse direction, so technically there are two rules of inference here. I will refer to both rules as (*Trading*). This *seamless transfer between program and pre-program has no analogue in Hoare logic*, and provides a major tactic for the derivation of programs from operational specifications.

Append Rule. Appending the same program γ to the program and the post-program preserves the validity of an operational triple. This is useful for appending new code into both the program and the post-program.

Substitution Rule. Since the definition of operational annotation refers only to the behavior of a program, it follows that one equivalent program can be replaced by another. This rule is useful for performing equivalence-preserving transformations, such as loop unwinding.

Table 1. Axioms and rules of inference

$[\alpha]\, P\, [\alpha; P]$ (*Sequence Axiom*)

$[\mathit{skip}]\, P\, [P]$ (*Empty Pre-program*)

$[\alpha]\, \mathit{skip}\, [\alpha]$ (*Empty Program*)

$[\alpha]\, P1; P2\, [\beta]$ iff $[\alpha; P1]\, P2\, [\beta]$ (*Trading*)

$$\frac{[\alpha]\, P\, [\beta]}{[\alpha]\, P; \gamma\, [\beta; \gamma]}$$ (*Append*)

$$\frac{\alpha \equiv \alpha' \quad P \equiv P' \quad \beta \equiv \beta' \quad [\alpha]\, P\, [\beta]}{[\alpha']\, P'\, [\beta']}$$ (*Substitution*)

$$\frac{\alpha \preceq \alpha' \quad [\alpha']\, P\, [\beta]}{[\alpha]\, P\, [\beta]}$$ (*Pre-program Strengthening*)

$$\frac{[\alpha]\, P\, [\beta'] \quad \beta' \preceq \beta}{[\alpha]\, P\, [\beta]}$$ (*Post-program Weakening*)

$$\frac{[\alpha]\, P1\, [\beta] \quad [\beta]\, P2\, [\gamma]}{[\alpha]\, P1; P2\, [\gamma]}$$ (*Sequential Composition*)

$$\frac{[\alpha']\, P\, [\alpha]}{[\alpha]\, \textbf{\textit{while}}(B)\ P\ \textbf{\textit{elihw}}\, [\beta]}\ \alpha' \cong (\alpha, B), \beta \cong (\alpha, \neg B)$$ (*While*)

$$\frac{[\alpha']\, P\, [\gamma; \alpha]}{[\alpha]\, \textbf{\textit{while}}(B)\ P\ \textbf{\textit{elihw}}\, [\beta]}\ \alpha' \cong (\alpha, B), \beta \cong (\alpha, \neg B)$$ (*While Consequence*)

$$\frac{[\alpha']\, P1\, [\beta] \qquad [\alpha'']\, P2\, [\beta]}{[\alpha]\, \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ P1\ \textbf{\textit{else}}\ P2\ \textbf{\textit{fi}}\, [\beta]}\ \alpha' \cong (\alpha, B), \alpha'' \cong (\alpha, \neg B)$$ (*If*)

$$\frac{[\alpha']\, P\, [\beta] \qquad \alpha'' \preceq \beta}{[\alpha]\, \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ P\ \textbf{\textit{fi}}\, [\beta]}\ \alpha' \cong (\alpha, B), \alpha'' \cong (\alpha, \neg B)$$ (*One-way If*)

Pre-program Strengthening. Reducing the set of post-states of the pre-program cannot invalidate an operational triple.

Post-program Weakening. Enlarging the set of post-states of the post-program cannot invalidate an operational triple.

Sequential Composition. The post-state set of β serves as the intermediate state-set in the execution of $P1; P2$: it characterizes the possible states after $P1$ executes and before $P2$ executes.

While Rule. Given $[\alpha]\, P\, [\alpha]$, I wish to conclude $[\alpha]\ \textbf{\textit{while}}(B)\ P\ \textbf{\textit{elihw}}\ [\beta]$ where β is a "conjunction" of α and $\neg B$, i.e., the post-states of β are those that are post-states of α, and also that satisfy assertion $\neg B$, the negation of the looping condition. Also, I wish to weaken the hypothesis of the rule from $[\alpha]\, P\, [\alpha]$ to $[\alpha']\, P\, [\alpha]$, where α' is a "conjunction" of α and B, i.e., the post-states of α' are those that are post-states of α and that also satisfy assertion B, the looping condition. I therefore define the "conjunction" of a program and an assertion as follows. Let α', α be programs and B a Boolean expression. Then define $\alpha' \cong (\alpha, B) \triangleq post(\alpha') = post(\alpha) \cap \{s \mid s(B) = tt\}$. Note that this definition does not produce a unique result, and so is really a relation rather than a mapping. The construction of α' is not straightforward, in general, for arbitrary assertions B. Fortunately, most looping conditions are simple, typically a loop counter reaching a limit. I therefore define the needed program α' by the semantic condition given above, and leave the problem of deriving α' from α and B to another occasion.

Given a while loop $\textbf{\textit{while}}(B)\ P\ \textbf{\textit{elihw}}$ and pre-program α, let α' be a program such that $\alpha' \cong (\alpha, B)$, and let β be a program such that $\beta \cong (\alpha, \neg B)$. The hypothesis of the rule is: execute α and restrict the set of post-states to those in which B holds. That is, have α' as a pre-program for the loop body P. First assume that B holds initially. Then, after P is executed, the total resulting effect must be the same as executing just α. So, α is a kind of "operational invariant". Given that this holds, and taking α as a pre-program for $\textbf{\textit{while}}(B)\ P\ \textbf{\textit{elihw}}$, then upon termination, we have α as a post-program. On the last iteration of the while loop, B is false, and the operational invariant α still holds. In case B is initially false, the while loop terminates immediately with no change of state. Hence α holds, since it held initially, and B is false. Hence in both cases I can assert β as a post-program for the ***while*** loop.

While Rule with Consequence. By applying Proposition 1 and (*Post-program Weakening*) to (*While*), I obtain (*While Consequence*), which states that the operational invariant can be a "suffix" of the actual post-program of the loop body. This is often convenient, in practice.

If Rule. Let α be the pre-program. Assume that execution of $P1$ with pre-program $\alpha' \cong (\alpha, B)$ leads to post-program β, and that execution of $P2$ with pre-program $\alpha'' \cong (\alpha, \neg B)$ also leads to post-program β. Then, execution of $\textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ P1\ \textbf{\textit{else}}\ P2\ \textbf{\textit{fi}}$ with pre-program α leads to post-program β.

One-Way If Rule. Assume that execution of $P1$ with pre-program $\alpha' \cong (\alpha, B)$ leads to post-program β. Also assume that any post-state of α is also a post-state of β. Then execution of $\textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ P1\ \textbf{\textit{fi}}$ with pre-program α always leads to post-program β.

A notable omission from the above rules is an assignment axiom. Letting P be $x := e$ in (*Sequence Axiom*), I obtain $[\alpha]\, x := e\, [\alpha; x := e]$. This can be regarded as the operational analogue of the Hoare Logic assignment axiom. The post-program $\alpha; x := e$ can then be further manipulated, e.g., by equivalence transformations, or by being a pre-program for the statement following $x := e$.

6 Example: In-Place List Reversal

I now illustrate the use of operational annotations to derive a correct algorithm for the in-place reversal of a linked list. The full version [1] also contains examples of deriving selection sort, Dijkstra's shortest path algorithm, and binary search tree node insertion, from operational specifications. (*Trading*) is used heavily in all these examples. Throughout, I use informal arguments for program equivalence, based on well-known transformations such as eliminating the empty program *skip*, and unwinding the last iteration of a ***for*** loop. Pre/post-programs are written in bold red italics, and regular programs are written in typewriter.

The input is a size $\ell + 1$ array n of objects of type `Node`, where $\ell \geqslant 3$, and indexed from 0 to $\ell - 1$. The last element is set to NIL and serves as a sentinel. `Node` is declared as follows: *class* `Node`$\{$`Node` p; other fields $\ldots\}$. Element i is referred to as n_i instead of $n[i]$, and contains a pointer $n_i.p$, and possibly other (omitted) fields. The use of this array is purely for specification purposes, so that I can construct the initial linked list from elements $n_0, \ldots, n_{\ell-1}$. An array also ensures that there is no aliasing: all elements are distinct, by construction. Array n is created by executing `Node`$[\,]\ n := new$ `Node`$[\ell + 1]; n.\ell :=$ NIL. The pre-program and post-program both start with code to declare `Node`, followed by the above line to create array n. I omit this code as including it would be repetitive and would add clutter.

I start by applying (*Empty Program*), which gives the following:

$i := [0 : \ell - 3];$
for $(j = 0$ ***to*** $\ell - 1)\ n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i + 1)\ n_j.p := n_{j-1}$ ***rof***;
$r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$
`skip`
$i := [0 : \ell - 3];$
for $(j = 0$ ***to*** $\ell - 1)\ n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i + 1)\ n_j.p := n_{j-1}$ ***rof***;
$r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$

The pre (and post) programs do three things: (1) construct the linked list by setting $n_j.p$ to point to the next node n_{j+1}, including the last "real" node $n_{\ell-1}$, which points to the sentinel n_ℓ, and (2) reverse part of the list, up to position $i+1$, by setting $n_j.p$ to point to the previous node n_{j-1}, for j from $i+1$ down to the second node n_1, and (3) maintain 3 pointers, into positions $i+1$, $i+2$, and $i+3$. The pre (post) programs are simple enough that I take their correctness for

granted. The topic of writing correct specifications is of course a major concern of software engineering [29]. One of the possible post-states of the pre (post) programs has $t = n_\ell$; this is the termination state for the overall algorithm, as we will see. Now unwind the last iteration of the second ***for*** loop of the pre-program. So, by (*Substitution*)

$i := [0 : \ell - 3]$;
for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i)$ $n_j.p := n_{j-1}$ ***rof***;
$n_{i+1}.p := n_i$;
$r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$
`skip`
$i := [0 : \ell - 3]$;
for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i + 1)$ $n_j.p := n_{j-1}$ ***rof***;
$r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$

Now introduce $r := n_i; s := n_{i+1}; t := n_{i+2}$ into the pre-program. Since r, s, t are subsequently overwritten, and not referenced in the interim, this preserves equivalence of the pre-program with its previous version. So, by (*Substitution*)

$i := [0 : \ell - 3]$;
for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i)$ $n_j.p := n_{j-1}$ ***rof***;
$r := n_i; s := n_{i+1}; t := n_{i+2}$;
$n_{i+1}.p := n_i$;
$r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$
`skip`
$i := [0 : \ell - 3]$;
for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i + 1)$ $n_j.p := n_{j-1}$; ***rof***;
$r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$

Now apply (*Trading*) to move the last two lines of the pre-program into the program, and remove the *skip* since it is no longer needed.

$i := [0 : \ell - 3]$;
for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i)$ $n_j.p := n_{j-1}$ ***rof***;
$r := n_i; s := n_{i+1}; t := n_{i+2}$
$\mathtt{n_{i+1}.p} := \mathtt{n_i}$;
$\mathtt{r} := \mathtt{n_{i+1}}; \mathtt{s} := \mathtt{n_{i+2}}; \mathtt{t} := \mathtt{n_{i+3}}$
$i := [0 : \ell - 3]$;
for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $i + 1)$ $n_j.p := n_{j-1}$ ***rof***;
$r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$

To avoid repetitive text, and to anticipate the development of the operational invariant for the while loop that goes through the linked list, I define the abbreviations

$inv \triangleq$
$\quad \textbf{\textit{for}}\ (j = 0\ \textbf{\textit{to}}\ \ell - 1)\ n_j.p := n_{j+1}\ \textbf{\textit{rof}};$
$\quad \textbf{\textit{for}}\ (j = 1\ \textbf{\textit{to}}\ i)\ n_j.p := n_{j-1}\ \textbf{\textit{rof}};$
$\quad r := n_i; s := n_{i+1}; t := n_{i+2}$

and

$inv' \triangleq$
$\quad \textbf{\textit{for}}\ (j = 0\ \textbf{\textit{to}}\ \ell - 1)\ n_j.p := n_{j+1}\ \textbf{\textit{rof}};$
$\quad \textbf{\textit{for}}\ (j = 1\ \textbf{\textit{to}}\ i)\ n_j.p := n_{j-1}\ \textbf{\textit{rof}};$
$\quad r := n_{i+1}; s := n_{i+2}; t := n_{i+3}$

Since $r := n_i; s := n_{i+1}$ immediately precedes $\mathtt{n_{i+1}.p} := \mathtt{n_i}$, I can replace $\mathtt{n_{i+1}.p} := \mathtt{n_i}$ by $\mathtt{s.p} := \mathtt{r}$ while retaining equivalence. So, by (*Substitution*)

$i := [0 : \ell - 3]; inv$
$\mathtt{s.p} := \mathtt{r};$
$\mathtt{r} := \mathtt{n_{i+1}}; \mathtt{s} := \mathtt{n_{i+2}}; \mathtt{t} := \mathtt{n_{i+3}}$
$i := [0 : \ell - 3]; inv'$

Since $s := n_{i+1}$ precedes $\mathtt{r} := \mathtt{n_{i+1}}$ and s is not modified in the interim, I can replace $\mathtt{r} := \mathtt{n_{i+1}}$ by $\mathtt{r} := \mathtt{s}$ while retaining equivalence. Likewise, since $t := n_{i+2}$ precedes $\mathtt{s} := \mathtt{n_{i+2}}$, and t is not modified in the interim, I can replace $\mathtt{s} := \mathtt{n_{i+2}}$ by $\mathtt{s} := \mathtt{t}$. So, by (*Substitution*) applied twice, I obtain

$i := [0 : \ell - 3]; inv$
$\mathtt{s.p} := \mathtt{r};$
$\mathtt{r} := \mathtt{s}; \mathtt{s} := \mathtt{t}; \mathtt{t} := \mathtt{n_{i+3}}$
$i := [0 : \ell - 3]; inv'$

From $\textbf{\textit{for}}\ (j = 0\ \textbf{\textit{to}}\ \ell - 1)\ n_j.p := n_{i+1}\ \textbf{\textit{rof}}$, I have $n_{i+2}.p := n_{i+3}$, and I observe that $n_{i+2}.p$ is not subsequently modified. Also I have $t := n_{i+2}$ occurring before the program, and t is not modified until $\mathtt{t} := \mathtt{n_{i+3}}$, so I can replace $\mathtt{t} := \mathtt{n_{i+3}}$ by $\mathtt{t} := \mathtt{t.p}$. Hence by (*Substitution*)

$i := [0 : \ell - 3]; inv$
$\mathtt{s.p} := \mathtt{r};$
$\mathtt{r} := \mathtt{s}; \mathtt{s} := \mathtt{t}; \mathtt{t} := \mathtt{t.p}$
$i := [0 : \ell - 3]; inv'$

Now apply (*While*) to obtain the complete program, while also incrementing the loop counter i at the end of the loop body.

$\textbf{\textit{for}}\ (j = 0\ \textbf{\textit{to}}\ \ell - 1)\ n_j.p := n_{j+1}\ \textbf{\textit{rof}}$
$\mathtt{r} := \mathtt{n_0}; \mathtt{s} := \mathtt{n_1}; \mathtt{t} := \mathtt{n_2}$

$i := 0; inv$
`while` $(\mathtt{i} \neq \ell - 2)$
  $i := [0 : \ell - 3]; inv$
  `s.p` := `r`;
  `r` := `s`; `s` := `t`; `t` := `t.p`
  $i := [0 : \ell - 3]; inv'$
  `i` := `i` + 1;
  $i := [1 : \ell - 2]; inv$
`elihw`
$i := \ell - 2; inv$

Upon termination, $i := \ell - 2$ expresses the negation of the looping condition, and so forms part of the post-program of the while loop, together with the operational invariant inv. I add `i` := `i` + 1 at the end of the loop body to re-establish the invariant. The justification for this is equivalence between $i := [0 : \ell - 3]; inv'$; `i` := `i` + 1 and $i := [1 : \ell - 2]; inv$, i.e., between (1) the range for i before incrementing, followed by the "modified" invariant inv', followed by the increment of i, and (2) the range for i after incrementing, followed by the invariant inv.

The loop terminates when t reaches the sentinel, i.e., when $t = n_\ell$. Since $t := n_{i+2}$ in the operational invariant, this requires $i = \ell - 2$. The post-program of the loop then gives ***for*** $(j = 1$ ***to*** $\ell - 2)$ $n_j.p := n_{j-1}$ ***rof***, which means that the last node's pointer is not set to the previous node, since the list ends at index $\ell - 1$. Hence we require a final assignment that is equivalent to $n_{\ell-1}.p := n_{\ell-2}$.

The post-program also gives $i := \ell - 2; r := n_i; s := n_{i+1}; t := n_{i+2}$, which yields $i := \ell - 2; r := n_{\ell-2}; s := n_{\ell-1}; t := n_\ell$. Hence the last assignment can be rendered as $s.p := r$. Also, the loop termination condition can be rewritten as $t \neq$ NIL, since t becomes NIL when it is assigned n_ℓ, which happens exactly when i becomes $\ell - 2$. Hence, using (*Substitution*) and (*Post-program Weakening*), I obtain

for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***
`r` := $\mathtt{n}_0$; `s` := $\mathtt{n}_1$; `t` := $\mathtt{n}_2$
$i := 0; inv$
`while` $(\mathtt{t} \neq$ NIL$)$
  $i := [0 : \ell - 3]; inv$
  `s.p` := `r`;
  `r` := `s`; `s` := `t`; `t` := `t.p`
  $i := [0 : \ell - 3]; inv'$
  `i` := `i` + 1;
  $i := [1 : \ell - 2]; inv$
`elihw`;
$i := \ell - 2; inv$
`s.p` := `r`
for $(j = 0$ ***to*** $\ell - 1)$ $n_j.p := n_{j+1}$ ***rof***;
for $(j = 1$ ***to*** $\ell - 1)$ $n_j.p := n_{j-1}$ ***rof***

Now, as desired, the "loop counter" i is no longer needed as a program variable, and can be converted to an auxiliary ("ghost") variable. The result is

```
for (j = 0 to ℓ − 1) n_j.p := n_{j+1} rof
r := n_0; s := n_1; t := n_2
i := 0; inv
while (t ≠ NIL)
      i := [0 : ℓ − 3]; inv
      s.p := r;
      r := s; s := t; t := t.p
      i := [1 : ℓ − 2]; inv
elihw;
i := ℓ − 2; inv
s.p := r
for (j = 0 to ℓ − 1) n_j.p := n_{j+1} rof;
for (j = 1 to ℓ − 1) n_j.p := n_{j−1} rof
```

Upon termination, `s` points to the head of the reversed list. The post-program is quite pleasing: it constructs the list, and then immediately reverses it!

7 Operational Annotations for Procedures

Let *pname* be a non-recursive procedure with parameter passing by value, and with body *pbody*. Let $\overline{a}$ denote a list of actual parameters, and let $\overline{f}$ denote a list of formal parameters. An actual parameter is either an object identifier or an expression over primitive types, and a formal parameter is either an object identifier or a primitive-type identifier.

Let r be an identifier matching the return type of *pname*, and not occurring in *pbody*. To avoid complex substitutions I assume that: (1) all local variables/objects in *pbody* do not have the same identifier as any variable/object in the calling context and (2) there is only a single return statement, which is the last statement of *pbody*. These assumptions are not fundamental; they can be removed at the price of somewhat more complex rules.

$$pname(\overline{a}) \;\equiv\; \overline{f} := \overline{a}; pbody[skip/\textbf{\textit{return}}] \qquad (Equiv\, Nonrecursive\, Void)$$

$$r := pname(\overline{a}) \;\equiv\; \overline{f} := \overline{a}; pbody[r := e/\textbf{\textit{return}}(e)] \qquad (Equiv\, Nonrecursive)$$

Equiv Nonrecursive Void handles calls where the returned value does not exist (void return type). *Equiv Nonrecursive* handles assignment of the returned value to r by textually replacing ***return***(e) by $r := e$ in the procedure body. Consider the linked list reversal algorithm above, packaged as a procedure:

```
Node Procedure listRev(Node h) {
      r := h; s := h.p; t := h.p.p;
      while (t ≠ NIL)
            s.p := r;
```

```
            r := s; s := t; t := t.p
        elihw;
        s.p := r;
        return(s)
    }
```

Now consider a call $v := listRev(\ell)$ where ℓ is the head of a linked list. By (*Equiv Nonrecursive*), I have:

```
v := listRev(ℓ) ≡ {
        h := ℓ;
        r := h; s := h.p; t := h.p.p;
        while (t ≠ NIL)
            s.p := r;
            r := s; s := t; t := t.p
        elihw;
        s.p := r;
        v := s
        }
```

This gives the correct effect for the procedure call $listRev(\ell)$, namely that v points to the head of the reversed list. In particular, both primitive and reference types (as parameters) are handled correctly, and need not be distinguished in the above rules.

For recursive procedures, an inductive proof method is needed, as usual. I use an inductive rule to establish the equivalence between a sequence of two procedure calls and a third procedure call. These correspond, respectively, to the pre-program, the program, and the post-program. The method is as follows:

1. Let α, P, β be recursive procedures which give, respectively, the pre-program, program, and post-program
2. To establish $\alpha; P \equiv \beta$ I proceed as follows:
   (a) Start with $\alpha; P$, replace the calls by their corresponding bodies, and then use a sequence of equivalence-preserving transformations to bring any recursive calls $\alpha'; P'$ next to each other
   (b) Use the inductive hypothesis for equivalence of the recursive calls $\alpha'; P' \equiv \beta'$ to replace $\alpha'; P'$ by β'
   (c) Use more equivalence-preserving transformations to show that the resulting program is equivalent to β

The appropriate rule of inference is as follows:

$$\frac{\alpha'; P' \equiv \beta' \ \vdash \ \alpha; P \equiv \beta}{\alpha; P \equiv \beta} \qquad (Equiv\ Recursive) \tag{1}$$

where $\vdash$ means "is deducible from". Thus, if we prove $\alpha; P \equiv \beta$ (pre-program followed by program is equivalent to post-program) by assuming $\alpha'; P' \equiv \beta'$ (a

recursive invocation of the pre-program followed by a recursive invocation of the program is equivalent to a recursive invocation of the post-program), then we conclude, by induction on recursive calls, that $\alpha; P \equiv \beta$.

The following example verifies the standard recursive algorithm for node insertion into a binary search tree (BST). Each node n consists of three fields: $n.key$ gives the key value for node n, $n.\ell$ points to the left child of n (if any), and $n.r$ points to the right child of n (if any). The constructor `Node(k)` returns a new node with key value `k` and null left and right child pointers. I assume that all key values in the tree are unique. For clarity, I retain red italics for specification code and black typewriter for program code. Procedure `insert(T,k)` gives the standard recursive algorithm for insertion of key `k` into a BST with root `T`.

```
insert(T,k)::
  if (T = NIL) T := new Node(k);
  else if (k < T.key) insert(T.ℓ,k);
  else insert(T.r,k);
```

To verify the correctness of `insert(T,k)`, I define two recursive procedures $ct(T,\psi)$ and $cti(T,\psi,k)$. $ct(T,\psi)$ takes a set ψ of key values, constructs a random binary search tree containing exactly these values, and sets T to point to the root of this tree. $x := select\ in\ \psi$ selects a random value in ψ and assigns it to x. $cti(T,\psi,k)$ takes a set ψ of key values and a key value $k \notin \psi$, constructs a random binary search tree which contains exactly the values in ψ together with the key k, and where k is a leaf node, and sets T to point to the root of this tree.

$ct(T,\psi)$::

if $(\psi = \emptyset)$ $T :=$ NIL;
else
    $x := select\ in\ \psi;$
    $\psi := \psi - x;$
    Node $T :=$ ***new Node***$(x);$
    $ct(T.\ell, \{y \mid y \in \psi \wedge y < x\});$
    $ct(T.r, \{y \mid y \in \psi \wedge y > x\})$

$cti(T,\psi,k)$::

if $(\psi = \emptyset)$ $T :=$ ***new Node***(k);
else
    $x := select\ in\ \psi;$
    $\psi := \psi - x;$
    Node $T :=$ ***new Node***$(x);$
    if $(k < x)$
        $cti(T.\ell, \{y \mid y \in \psi \wedge y < x\}, k);$
        $ct(T.r, \{y \mid y \in \psi \wedge y > x\})$

else
$ct(T.\ell, \{y \mid y \in \psi \wedge y < x\})$
$cti(T.r, \{y \mid y \in \psi \wedge y > x\}, k)$

I now verify

$$[ct(T, \varphi)]\ \texttt{insert(T, k)}\ [ct(T, \varphi \cup k)]. \qquad (*)$$

That is, the pre-program creates a random BST with key values in φ and sets T to the root, and the post-program creates a random BST with key values in $\varphi \cup k$ and sets T to the root. Hence, the above operational triple states that the result of `insert(T, k)` is to insert key value `k` into the BST rooted at `T`. I first establish

$$cti(T, \varphi, k) \preceq ct(T, \varphi \cup k). \qquad (a)$$

Intuitively, this follows since $cti(T, \varphi, k)$ constructs a BST with key values in $\varphi \cup k$, and where k must be a leaf node, while $ct(T, \varphi \cup k)$ constructs a BST with key values in $\varphi \cup k$, with no constraint of where k can occur. A formal proof proceeds by induction on the length of an arbitrary execution π of $cti(T, \varphi, k)$, which shows that π is also a possible execution of $ct(T, \varphi \cup k)$. The details are straightforward and are omitted. In the sequel, I show that

$$[ct(T, \varphi)]\ \texttt{insert(T, k)}\ [cti(T, \varphi, k)] \qquad (b)$$

is valid. From (a,b) and (*Post-program Weakening*), I conclude that (*) is valid, as desired. To establish (b), I show

$$ct(T, \varphi); \texttt{insert(T, k)} \equiv cti(T, \varphi, k). \qquad (c)$$

from which (b) follows immediately by Definitions 2 and 4.

I establish (c) by using induction on recursive calls. I replace the above calls by the corresponding procedure bodies, and then assume as inductive hypothesis (c) as applied to the recursive calls within the bodies. This is similar to the Hoare logic inference rule for partial correctness of recursive procedures [7].

To apply the inductive hypothesis, I take $ct(T, \varphi)$; `insert(T, k)`, replace each call by the corresponding procedure body, and then "interleave" the procedure bodies using commutativity of statements, which preserves equivalence. Then I bring the recursive calls to ct and to `insert` together, so that the inductive hypothesis applies to their sequential composition, which is then replaced by the equivalent recursive call to cti. This gives a procedure body that corresponds to a call of cti, which completes the equivalence proof. The result is as follows (see [1] for full details):

if $(\varphi = \emptyset)$ `T := new Node(k)`
else
$x :=$ *select in* φ;
$\varphi := \varphi - x$;
Node $T :=$ ***new Node***(x);

```
if (k < T.val)
      cti(T.ℓ, {y | y ∈ φ ∧ y < x}, k);
      ct(T.r, {y | y ∈ φ ∧ y > x});
else
      ct(T.ℓ, {y | y ∈ φ ∧ y < x});
      cti(T.r, {y | y ∈ φ ∧ y > x}, k)
```

and is the procedure body corresponding to the call $cti(T, \varphi, k)$. By (*Equiv Recursive*), I conclude $ct(T, \varphi)$; `insert(T,k)` $\equiv cti(T, \varphi, k)$, which is (c) above. This completes the proof.

8 Conclusions

I presented a new method for verifying the correctness of sequential programs. The method does not use assertions, but rather pre-/post-programs, each of which define a set of post-states, and can thus replace an assertion, which also defines a set of states, namely the states that satisfy it. Since pre-/post-programs are not executed, they can be inefficient, and can refer to any mathematically well-defined quantity, e.g., shortest path distances in a directed graph. I illustrated my method using as examples in-place list-reversal and an abbreviated BST node insertion. The full version of the paper [1] also presents derivations of selection sort and Dijkstra's shortest path algorithm, as well as giving full details for BST node insertion.

My approach is, to my knowledge, the first which uses *purely* operational specifications to verify the correctness of sequential programs, as opposed to pre- and post-conditions and invariants in a logic such as Floyd-Hoare logic and separation logic, or axioms and signatures in algebraic specifications [30]. The use of operational specifications is of course well-established for the specification and verification of concurrent programs. The process-algebra approach [8,16,17] starts with a specification written in a process algebra formalism such as CSP, CCS, or the Pi-calculus, and then refines it into an implementation. Equivalence of the implementation and specification is established by showing a bisimulation [16,21] between the two. The I/O Automata approach [14] starts with a specification given as a single "global property automaton" and shows that a distributed/concurrent implementation respects the global property automaton by establishing a simulation relation [15] from the implementation to the specification. Future work includes more examples and case studies, and in particular examples with pointer-based data structures. I am also investigating program refinement in the context of operational annotations, and the extension of the operational annotations approach to the verification of concurrent programs.

References

1. Attie, P.C.: Operational annotations: A new method for sequential program verification. CoRR abs/2102.06727 (2021). https://arxiv.org/abs/2102.06727

2. Back, R., von Wright, J.: Refinement Calculus - A Systematic Introduction. Graduate Texts in Computer Science. Springer (1998). https://doi.org/10.1007/978-1-4612-1674-2
3. Ciobâcă, Ş, Lucanu, D., Rusu, V., Roşu, G.: A language-independent proof system for full program equivalence. Formal Aspects Comput. **28**(3), 469–497 (2016). https://doi.org/10.1007/s00165-016-0361-7
4. Crole, R.L., Gordon, A.D.: Relating operational and denotational semantics for input/output effects. Math. Struct. Comput. Sci. **9**(2), 125–158 (1999). http://journals.cambridge.org/action/displayAbstract?aid=44797
5. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
6. Floyd, R.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics, pp. 19–32. American Mathematical Society (1967)
7. Francez, N.: Program verification. Addison-Wesley, International computer science series (1992)
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
9. Hoare, C.A.R., et al.: Laws of programming. Commun. ACM **30**(8), 672–686 (1987). https://doi.org/10.1145/27651.27653
10. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580, 583 (1969)
11. Hoare, T.: Laws of programming: the algebraic unification of theories of concurrency. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 1–6. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_1
12. Jacobs, B.: Dijkstra and Hoare monads in monadic computation. Theor. Comput. Sci. **604**, 30–45 (2015). https://doi.org/10.1016/j.tcs.2015.03.020
13. Lucanu, D., Rusu, V.: Program equivalence by circular reasoning. Formal Aspects Comput. **27**(4), 701–726 (2014). https://doi.org/10.1007/s00165-014-0319-6
14. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI-Quarterly **2**(3), 219–246 (1989), centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988
15. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. Untimed systems. Inf. Comput. **121**(2), 214–233 (1995). https://doi.org/10.1006/inco.1995.1134
16. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
17. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
18. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, 5–8 June, 1989, pp. 14–23. IEEE Computer Society (1989). https://doi.org/10.1109/LICS.1989.39155
19. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
20. Morgan, C.: Programming from specifications, 2nd edn. Prentice Hall International series in computer science, Prentice Hall (1994)
21. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981). https://doi.org/10.1007/BFb0017309

22. Pitts, A.M.: Operational semantics and program equivalence. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 378–412. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45699-6_8
23. Pitts, A.M., Stark, I.D.B.: Observable properties of higher order functions that dynamically create local names, or: What's new? In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 122–141. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57182-5_8
24. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebraic Methods Program. **60–61**, 17–139 (2004)
25. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS 2002, pp. 55–74. IEEE Computer Society, Washington, DC (2002). http://dl.acm.org/citation.cfm?id=645683.664578
26. Schmidt, D.A.: Programming language semantics. In: Gonzalez, T.F., Diaz-Herrera, J., Tucker, A. (eds.) Computing Handbook, Third Edition: Computer Science and Software Engineering, pp. 69: 1–19. CRC Press (2014)
27. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F. In: Bodik, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 256–270. ACM (2016). https://doi.org/10.1145/2837614.2837655
28. Swierstra, W.: A hoare logic for the state monad. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 440–451. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_30
29. Wing, J.M.: Hints to specifiers. Teaching and learning formal methods, pp. 57–78 (1995)
30. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 675–788. Elsevier and MIT Press (1990). https://doi.org/10.1016/b978-0-444-88074-1.50018-4