Compositional Non-Termination Proving

AZALEA RAAD, Imperial College London, UK
JULIEN VANEGUE, Bloomberg, USA
PETER O’HEARN, University College London and Lacework, UK

Program termination is a classic non-safety property that cannot in general be witnessed by a finite trace. This makes testing for non-termination challenging, and also makes it a natural target for symbolic proof. To confirm that non-termination is a practical and not theoretical problem, we provide a manual analysis of CVE’s due to non-termination, corresponding to security issues such as DOS vulnerabilities, finding 916 since 2000. Discovering non-termination is an under-approximate problem. We thus present UNTer, a sound and complete under-approximate logic for proving non-termination. We then extend UNTer with separation logic and develop UNTerSL for programs that manipulate the heap. UNTerSL yields a compositional proof method, which is amenable to automation via program analysis tools based on under-approximation and bi-abduction. We briefly describe a prototype tool, Pulse∞, under development, which extends the compositional the Pulse analyser from Facebook.

Additional Key Words and Phrases: Divergence, non-termination, under-approximation, incorrectness logic

1 INTRODUCTION

Why Prove Non-termination? Non-termination is a fundamental problem in computer science, dating back to the halting problem. Assuming an unbounded memory or tape, neither it nor its complement is recursively enumerable, making it difficult to approach using testing. This makes non-termination an attractive target for symbolic proof techniques.

Apart from its fundamental nature, one can also ask: is non-termination a practical problem? To understand this better we carried out a manual evaluation of CVE’s, security bugs such as denial of service which are due to non-termination. We found 916 such CVE’s between 2000 and 2022. Sometimes, for ongoing computations such as operating systems, potential non-termination is desirable and unavoidable. But, we may conclude that import and do have an effect.

Interestingly, we did not detect any reduction in non-termination CVE’s during this period. For example, we found 4 such bugs from 2000 and 28 from 2022. We stress that our manual approach might have missed some non-termination CVE’s, there is more code in 2022 than in 2000, and the classification of non-termination CVE’s might be non-uniform. This data, however, motivated our work on the science and engineering of tools for detecting non-termination bugs.

Why Compositional? A compositional analysis is one where the analysis result of a composite program is computed from those of its constituent parts [5]. Compositional analysis enables program analysis to be deployed as part of a code review process, where code snippets in a pull request are analysed without the need to re-analyse the entire program (or even to have an entire program, which might not yet exist). A case study from Facebook [14] describes how deploying a compositional static analysis tool on pull requests achieved a 70% fix rate, while the same analysis had a near 0% fix rate for a batch deployment (where a list of bugs is given outside of code review). This illustrates how a deployment of static analysis that meets programmers in their workflows can have considerable advantages over ones that ask them to leave their flow. (See the Facebook article [14] and a related article from Google [26] for more information.)

It stands to reason that if an accurate non-termination prover is developed which is fast enough to be deployed at pull-request time, then it would have the potential to have more non-termination bugs fixed, early. We will not in this paper go so far as setting up an industrial deployment of...
non-termination proving in the CICD system of a company, but we take the Facebook/Google experience referenced above as motivation for our scientific goals: to establish a compositional proof method together with an algorithm which allow for automatic compositional program analysis, and initial experiments to probe its feasibility.

Our Approach. Proving non-termination is an under-approximation problem as the aim is to establish the existence of non-terminating executions. Therefore, for compositional reasoning it is natural to consider a formalism akin to incorrectness logic (IL) [23], which brings the compositional nature of Hoare logic to bug proving. It turns out the form of under-approximation we need is a reversed form of that in IL, based on what is called the ‘backwards under-approximate triple’ by Möller et al. [22] and the ‘total Hoare triple’ by de Vries and Koutavas [13].

The backwards under-approximate (BUA) triple \( \vdash_B [p] \ C \ [ok : q] \) denotes that \( p \) is a subset of the states from which \( q \) can be reached executing \( C \). That is, from any state in \( p \) it is possible to reach some state in \( q \) by executing \( C \). This triple is forwards in terms of reachability, but backwards in terms of under-approximation (mirroring IL): \( q \) under-approximates the weakest possible precondition, \( \text{wpp} \), of \( C \) on \( q: p \subseteq \text{wpp}(C, q) \). Here, \( \text{wpp} \) is the inverse image of the \( C \) (relational) semantics, obtained by running Dijkstra’s strongest post-condition on the reversal of \( C \).

To this form of under-approximate (UA) triples we add another, for divergence. Specifically, we develop under-approximate non-termination logic (UNTer), where we write \( \vdash [p] \ C \ [\infty] \) to denote that every state in \( p \) leads to a divergent (infinite) execution via \( C \). Note that this does not state that every execution diverges; rather, each pre-state leads to some divergent execution. Given these triples we can state a proof rule for divergence as follows:

\[
\begin{align*}
\vdash_B [p \land B] & \ C \ [ok : p \land B] \\
\hline
\vdash_B [p \land B] & \text{while (B) } C \ [\infty]
\end{align*}
\]

The idea behind this rule is very simple. As \( p \land B \) holds initially, we know that after one loop iteration we can get to a state where \( p \land B \) continues to hold because of the triple in the premise. And in that case we can take one more step, ad infinitum.

This proof method is related intuitively to a method of non-termination testing whereby one looks for a concrete state to which a loop returns: this would witness divergence as one can get back to the same state again. As a testing method this approach is incomplete, in the presence of unbounded resources (e.g. a Turing machine tape) which gives rise to infinitely many states: then it is possible to diverge with returning to the same state twice. But the logical proof method uses a logical assertion and not a concrete state, and is in fact complete for proving non-termination as we prove later (take \( p \) to be the set of all states that lead to divergence).

The proof method is also related to the idea of ‘recurrence sets’ in a fundamental paper of Gupta et al. [17]. We say more on the relation to their and other work in §9.

Our aim is to automate divergence proof rules such as that above. There are several key observations in our approach. First, and remarkably, if we apply the strategy used commonly in abstract interpretation, namely iterating the abstract semantics of loops until we reach a fixpoint, then will have proven non-termination of a loop when a fixpoint is reached. In abstract interpretation this would not imply divergence, but with our under-approximate UNTer logic it does. However, while we can employ the usual method of fixpoint iteration, since not all loops diverge, we additionally need a way to stop the analysis before a fixpoint is reached. It turns out that we can employ similar techniques to IL and bounded model checking, by simply stopping after some fixed number of iterations even when we do not have a fixpoint. This flexibility is not available in Hoare logic, or in over-approximate abstract interpretation, where stopping early is unsound.
Second, by detailing the relationship to the original IL we reveal additional possibilities for
automation. Indeed, the BUA proof system is almost the same as that of IL, with the difference
limited to the rule of consequence (see §3, §4). The use of the backwards predicate transformer
wpp perhaps suggests to attempt a backwards program analysis, at least for a whole-program
analysis: given a post, such an analysis would compute an under-approximation of backwards
reachability at each program point; in a sense, the mirror image of Floyd’s method of calculating over-
approximations for forwards reachability. However, a forwards-running analysis is also possible, as
long as we abduce preconditions as we go forwards: this semantics calculates a collection of triples
at each program point, connecting procedure-entry to the program point. In addition to furnishing a
compositional inter-procedural analysis, abduction is necessary here: there is no forwards predicate
transformer semantics, evidenced by the fact that for some programs C and pre-conditions p there
is no post-condition delivering a valid triple ⊢ p \rightarrow [C \rightarrow ok : ??].

The third key point for automation is that the close connection between the BUA and original
IL proof theories suggests a method of automation that leverages separation logic [18], and which
is obtained by small changes and a fundamental addition to the existing Pulse program analyser
[19] from Facebook. We observe that Pulse uses a restricted version of the rule of consequence,
making it compatible both with BUA and IL triples. We thus develop UNTerSL as an extension of
UNTer (with divergent triples) with separation logic. We then extend Pulse with divergent triples
and develop Pulse∞, a prototype compositional non-termination prover underpinned by UNTerSL.

**Contributions and Outline.** Our contributions in this paper are as follows.

§2 We provide a manual classification of CVE’s related to non-termination, providing data to
go with existing anecdata, confirming the real-world prevalence of non-termination bugs
important enough to be judged as critical security issues.

§3 We present an intuitive overview of BUA and IL reasoning, and describe how we extend
them to reason about non-termination.

§4 We present UNTer as a BUA proof system and extend it to account for non-termination,
yielding a compositional proof method.

§5 We present several examples of divergence and show how we can detect them using UNTer.

§6 We present the semantic model of UNTer and show that it is sound and complete.

§7 We develop UNTerSL by extending UNTer with separation logic for heap reasoning.

§8 We observe that an existing under-approximate reasoning tool, Pulse, can be simply extended
to provide a compositional, incremental prover for non-termination, Pulse∞: we outline our
prototype implementation of Pulse∞ which is in progress.

§9 We discuss the related work and future work.

**Additional Material.** The proofs of all stated theorems in the paper are given in the accompanying technical appendix.

2 DIVERGENCE VULNERABILITIES

Divergence bugs are widespread across a number of programming languages. We present several exam-

ples taken from the Common Vulnerabilities and Ex-

posures (CVE) database and categorize them along
common cases of vulnerabilities – see Fig. 1 for the
prevalence of divergence bugs. We focus on control-
flow-related divergent behaviours brought about on
certain inputs. In particular, we focus on capturing

![Fig. 1. Vulnerability trend for divergence bugs](image-url)
behaviours where termination is not intended (unlike interactive programs whose non-termination is expected and induced from an infinite message loop treating streams of incoming input requests), and guarantee that our approach focuses on detecting the most widespread vulnerability classes in publicly available code. We have selected a number of bugs that show a wide cross-section of programming languages and control-flow conditions.

**Infinite Loops.** Recursive implementations are common in parsers. In some cases, the loop condition is driven by the value of an integer variable (e.g. remaining stream bytes to be read), which can be dynamically set within the parsing loop as the parser reads the input. If the decrement value of such variable in an iteration is set to 0, the loop makes no more progress leading to an unintended divergence. Specifically, when a parsing sub-function $f$ is called to treat a sub-case of input data type, if $f$ returns 0, then the loop makes no progress reading input. Such an example was found in the popular Wireshark network analyser, leading to CVE-2022-3190 (see §G.1).

**Infinite Recursion.** Infinite recursion bugs are one of the main sources of divergence. Infinite recursion bugs are well-known to parser developers when the recursive parsing function allows input variable expansion or other generative capability, such that when the newly generated input after expanding variables is parsed through a recursive call, the number of subsequently needed recursive calls remains non-null. Such a case was seen in the widely used Log4j logging library for Java programs, leading to CVE 2021-45105 (see §G.2).

**Out-of-Order Transition Divergence.** Unintended divergence can result from a loop or recursive call to a parsing function where certain input values or record data types are expected to be treated in a certain order, and an out-of-order encoding results in an infinite cycle. In certain cases, special input tag types are intended to be found at certain parsing stages as to disallow spurious transitions. Such a vulnerability was discovered in the GraphQL language interpreter, where the string type name can be encoded in the input such that the parsing handler calls itself repeatedly (see §G.3 for an example vulnerability affecting Go programs).

**Zero-Sized Input Divergence.** Container data structures (e.g. lists or vectors) are typically implemented with access primitives where adding or removing elements can be achieved independently of the current number of elements in the container. This is done by maintaining a meta-data size field. When such data structures are implemented with linear memory access in mind, an additional size field is needed to ascertain the size of an element in the data structure. Whether such element is of a fixed or variable size, an element with zero size can lead to a container iterator that diverges when traversing the structure without making progress. Such a problem was identified in the Linux kernel, leading to CVE-2020-25641 and was fixed in Linux kernel version 3.13 (see §G.4).

**Offset-Encoded Divergence.** In parser programs it is sometimes possible for the input to describe the actual input offset at which the data object is found. When such input offset indirection occurs, a parsing loop or recursive function can diverge by returning to previously parsed input in a way that will redo previously completed work and diverge. An example of this bug can be found in the popular graphic software Blender, written in C. Additional state would be required to ensure that the current input offset is restored after such out-of-band element is read (see §G.5).

**Exception-Induced Divergence.** Some parser implementations use exceptions to treat special or error cases where a recovery logic must be encoded in a catch or except block. Exception-induced spurious transitions can then be encoded such that the induction variable is never increment-ed/decremented, leading to divergence. A particular example of such vulnerability can be found
in the Sklearn industry-standard library for machine learning and data analysis in Python, where a convergence-based discretisation algorithm can be made to never terminate if the exceptional execution path fails to break from the appropriate number of loop nesting levels (see §G.6).

**Algebraic Divergence.** Divergence bugs can be found in mathematical software, where specific algebraic conditions are expected on the input to reach a fixpoint in an iterative or recursive function. The OpenSSL cryptographic library contains such code, where a modular square root implementation for an elliptic curve group expects the residue of the recursive operation to reach value 1 eventually, but invalid input parameters fail to meet this condition, leading to CVE-2022-0778. This vulnerability allowed remote SSL/TLS connections to get stuck in an infinite loop (see §G.7). This example illustrates that even security code can be vulnerable to divergence bugs!

### 3 Overview

**Incorrectness Logic and Under-Approximate Reasoning.** As Godefroid [16] argues, the main value of analysis tools lies in the discovery of bugs, not in the proof of program correctness. A bug presented to a developer is often a more convincing utility of a tool than a correctness proof, which is often carried out under certain assumptions that may not hold. This is evidenced by the recent trend in under-approximate reasoning techniques [23–25] and their significant success at finding bugs on an industrial scale [19, 4]. Specifically, Incorrectness logic (IL) [23] was recently introduced as an under-approximate formal foundation for bug detection. It was later extended to enable compositional bug detection in heap-manipulating programs [24], and to support concurrency [25]. IL and its later extensions are instances of under-approximate reasoning and are associated with no-false-positives theorems, ensuring that all bugs identified by them are true positives.

Intuitively, the under-approximate nature of IL stems from considering a subset of program behaviours. More concretely, given a program C whose behaviours (traces) is given by the set $S$, IL reasoning considers a subset (under-approximated) $S_u \subseteq S$ of the C behaviours. This makes IL ideally suited for bug-detection as it guarantees no-false-positives: if one detects a bug in the smaller set $S_u$, then the bug is also guaranteed to be in $S$ and thus exhibited by C. This is in contrast to over-approximate reasoning techniques such as Hoare logic, where one considers a superset (over-approximated) set $S_o \supseteq S$ of C behaviours, making them ideal for verification (as they guarantee no false negatives): if one can show that the larger set $S_o$ contains only correct behaviours, then the smaller set $S$ also contains correct behaviours only.

An IL triple, also referred to as a forward, under-approximate (FUA) triple, is of the form $\vdash F \: [p] \: C \: [e : q]$, where F hints at its forwards under-approximation, denoting that $q$ is a subset of program behaviours when C is run (forward) from the states in $p$. In other words, an FUA triple describes backward reachability: every post-state in $q$ is reachable by running C forward on some pre-state in $p$. The $e$ denotes an exit condition and may be either ok, to denote a normal execution or er to denote an erroneous execution. For instance, executing an explicit error statement (e.g. assert(false)) terminates erroneously and the underlying states are unchanged; this is given by the FUA triple $\vdash F \: [p] \: \text{error} \: [er : p]$. The under-approximate nature of FUA triples is best illustrated by their rules for reasoning about branches and loops. To show that a behaviour is possible when executing $C_1 + C_2$ (where $+$ denotes non-deterministic choice), it is sufficient to show the behaviour is possible when executing one of the branches, i.e. executing $C_i$ for some (rather than all) $i \in \{1, 2\}$, as shown in \texttt{Choice}F below (left). Similarly, to show a behaviour is possible when executing $C^*$ (where $C^*$ denotes a non-deterministic loop, executing C for zero or more iterations), it suffices to show it is possible when executing C for a particular number $n \in \mathbb{N}$ of iterations, as shown in \texttt{Loop}F below.
(right), where \( C^n \) denotes executing \( C \) for \( n \) times.

\[
\text{CHOICEF} \\
\text{\( \vdash_F [p] C_i [\epsilon : q] \) for some \( i \in \{1, 2\} \)} \\
\text{\( \vdash_F [p] C_1 + C_2 [\epsilon : q] \)}
\]

\[
\text{LOOPF} \\
\text{\( \vdash_F [p] C^n [\epsilon : q] \) for some \( n \in \mathbb{N} \)} \\
\text{\( \vdash_F [p] C^* [\epsilon : q] \)}
\]

**Non-termination and Under-Approximate Reasoning.** Existing literature includes a large body of work \([12, 15, 21, 2, 10, 3, 9, 6]\) on termination analysis, proving that a program \( C \) always terminates by showing that all traces of \( C \) terminate for all given inputs. Showing that a program \( C \) terminates is compatible with over-approximate reasoning frameworks. Specifically, when the traces of \( C \) are given by the set \( S \), showing that all traces in a larger set \( S_u \supseteq S \) terminate is sufficient for showing that all traces in \( S \) terminate. Showing termination is difficult in the presence of loops. In particular, to show that a loop \( L \) terminates typically involves the challenging task of establishing a loop invariant as well as a well-founded measure (a.k.a. a ranking function) that is decreased after each iteration \([12, 15]\). Establishing such invariants and measures is far from straightforward and typically involves reasoning about ordinal (rather than cardinal) numbers.

Showing that a program \( C \) does not terminate is compatible with under-approximate reasoning frameworks: when the traces (behaviours) of \( C \) are given by the set \( S \), showing that the traces in a smaller (under-approximate), possibly singleton, set \( S_u \subseteq S \) do not terminate is sufficient for showing that \( C \) does not terminate.

Inspired by the success of under-approximate analysis techniques and their industrial application of detecting bugs at scale, we develop under-approximate, non-termination logic (UNTer) as the first formal, under-approximate foundation for detecting non-termination bugs. As with existing under-approximate techniques, UNTer is associated with a no-false-positives theorem, ensuring that all non-termination bugs identified are true positives. More concretely, UNTer enables deriving under-approximate, divergent triples of the form \([p] C [\infty]\), stating that starting from the states in \( p \) program \( C \) has divergent (non-terminating) traces. Note that \([p] C [\infty]\) does not state that \( C \) never terminates (i.e. that all traces of \( C \) are divergent), but rather that it is possible for \( C \) not to terminate (i.e. some traces of \( C \) are divergent). For instance, given the program \( C \triangleq \text{skip} + (\text{while} (true) \text{ skip}) \), the triple \([true] C [\infty]\) is valid, since starting from any state (in \( true \)) \( C \) can always diverge by taking the right branch, even though taking the left branch would immediately lead to termination.

**Divergent Triples and FUA Triples.** As in the existing formal systems for reasoning about programs (be they over- or under-approximate), we should ideally reason about non-termination in a compositional fashion. For instance, given \( C_L \triangleq x := 1 \); while \((x > 0) \texttt{++}\) and an arbitrary initial value \( u \), to show that the triple \([x = u] C [\infty]\) holds (i.e. \( C_L \) does not terminate starting from states satisfying \( x = u \)), we should ideally show that 1) running \( x := 1 \) on states in which \( x = u \) terminates and modifies the states to those where \( x = 1 \); and 2) running while \((x > 0) \texttt{++}\) on states where \( x = 1 \) diverges, i.e. \([x = 1]\) while \((x > 0) \texttt{++}\) \([\infty]\). To do (1), we need to reason about non-divergent (terminating) program executions in an under-approximate fashion. At first glance, this seems an ideal job for FUA triples as they under-approximate reachable program behaviours upon termination; as such, to establish (1), we could simply show \( \vdash_F [x = u] x := 1 [ok : x = 1] \).

A key feature of our UNTer framework is proof rules for establishing when a loop does not terminate. As a first naive attempt, we can propose the LoopBad rule below (left), stating that if initially the while condition \( B \) holds, and executing one iteration of the loop body \( C \) starting from \( p \) leaves the states \( (p) \) and the loop condition \( B \) unchanged, then while \( (B) \) diverges.

\[
\text{LoopBad} \\
\text{\( \vdash_F [p \land B] C [ok : p \land B] \)} \\
\text{\( [p \land B] \) while \((B) \) C \([\infty]\)}
\]

\[
\text{LoopFix} \\
\text{\( \vdash_B [p \land B] C [ok : p \land B] \)} \\
\text{\( [p \land B] \) while \((B) \) C \([\infty]\)}
\]
On closer inspection, however, this rule is unsound. Consider the program while \((x > 0)\) \(x\ldots\); this program always terminates regardless of the value of \(x\) (for non-positive values the loop is never entered; positive values are eventually decremented to zero). As such, the triple \([x > 0]\) while \((x > 0)\) \(x\ldots[\infty]\) is invalid. Nevertheless, we can derive it using LoopBad by showing \(\mathbb{F} \ [x > 0] \ x\ldots[ok : x > 0]\). Specifically, the \(\mathbb{F} \ [x > 0] \ x\ldots[ok : x > 0]\) triple stipulates that every post-state in \(x > 0)\) be reachable from some pre-state in \(x > 0\), which is indeed the case. More concretely, consider an arbitrary post-state \(s_q \in x > 0\) and let \(s_q(x) = o\) (i.e. \(x\) holds value \(o\) in \(s_q\)) for some \(o > 0\). State \(s_q\) is then reachable by running \(x\ldots\) on a state \(s_p = s_q[x \mapsto o+1]\) and \(s_p \in x > 0\) (as \(o > 0\)).

### Backward Under-Approximate Triples

Intuitively, the problem lies in the backward reachability of FUA triples: it stipulates that each post-state be reachable from some pre-state, which does not necessarily lead to divergence. In other words, having a backward chain of \(C\) executions from \(p \land B\) to \(p \land B\) does not yield an infinite execution. Instead, we need a forward chain of \(C\) executions from \(p \land B\) to \(p \land B\), as we can then repeat this execution forward ad infinitum. This is captured in the LoopFix rule above (right), where a backward, under-approximate (BUA) triple \(\mathbb{B} \ [p] \ C\ [e : q]\) states that every pre-state in \(p\) reaches some post-state in \(q\) by executing \(C\). Therefore, if we show that each iteration of the loop body transitions each pre-state in \(p \land B\) to some post-state also in \(p \land B\), then we can repeat this transition infinitely, leading to divergence. Note that in the above example, we cannot show \(\mathbb{B} \ [x > 0] \ x\ldots[ok : x > 0]\) (unlike the \(\mathbb{F}\) variant): given state \(s_p \in x > 0\) with \(s_p(x) = 1\), running \(x\ldots\) on \(s_p\) yields a state \(s_q = s_p[x \mapsto 0]\), which is not in \(x > 0\).

As such, using LoopFix, we cannot derive the invalid triple \([x > 0]\) while \((x > 0)\) \(x\ldots[\infty]\). Note that while BUA triples describe forward reachability, they denote backward under-approximation: \(\mathbb{P} \subseteq \text{wpp}(C, q)\), where \(\text{wpp}(C, q)\) denotes running \(C\) backwards from \(q\). That is, BUA triples mirror FUA ones (which describe backward reachability but forward under-approximation).

In order to present our divergence proof rules in a compositional fashion, we thus use BUA triples to describe normal, terminating executions. For instance, in order to show that \(C_1; C_2\) does not terminate starting from \(p\), we can show either \(C_1\) does not terminate starting from \(p\) (i.e. \([p]\ C_1 [\infty]\)), or \(C_1\) terminates normally transforming the states to \(q\), and \(C_2\) does not terminate starting from \(q\) (i.e. \(\mathbb{B} \ [p] \ C_1 [ok : q] \text{ and } [q] C_2 [\infty]\)). This is captured by the \text{Div-Seq1} \text{ and Div-Seq2} rules in Fig. 3 (§4), where we present our full set of proof rules for detecting divergence.

### Forward versus Backward Under-Approximate Triples

As with FUA triples, BUA triples are also inherently under-approximate. Most notably, as we show in §4, the BUA rules for reasoning about branches and loops are identical to their FUA counterparts; i.e. the \(\mathbb{F}\) in \text{ChoiceF} and LoopF above can simply be replaced with \(\mathbb{B}\) (see Fig. 2). Indeed, almost all FUA and BUA proof rules coincide, and the only difference between FUA and BUA rules lie in their associated rules of consequence, namely the \text{ConsF} (for FUA) and \text{ConsB} (for BUA) rules in Fig. 2 (p. 11). However, as we describe shortly, in the practical context of industrially-deployed (under-approximate) bug detection tools such as Pulse [19], it is straightforward to reconcile this difference between FUA and BUA and to develop a unified, under-approximate reasoning framework.

The main application of the FUA rule of consequence, \text{ConsF}, is in conjunction with the rule of disjunction, \text{Disj} in Fig. 2 (p. 11). More concretely, when a given program contains multiple branches, thanks to the \text{ChoiceF} rule, we can analyse each branch (and not necessarily all branches) in isolation and generate a separate triple. Subsequently, we can merge them into a single triple using \text{Disj}. However, when there are many branches (and subsequently many disjuncts in the pre- and post-states), we can simply use \text{ConsF} to drop some of the disjuncts in the post-states. (Note that using \text{ConsB} analogously allows us to drop some of the disjuncts in the pre-states.)
However, as our conversations with the lead engineer behind Pulse have revealed, in the practical setting of such tools this scenario rarely arises, and it is handled differently when it does. Specifically, different triples of a program are not merged very often, as it is simpler and more efficient to keep them separate. Second, when triples are merged, they are done so in a fashion that additionally tracks the correspondence between the disjuncts in the pre- and post-states. Specifically, note that the \texttt{Disj} rule is lossy: while in its premise we know that the post-states in \(q_1\) (resp. \(q_2\)) are reached from the pre-states in \(p_1\) (resp. \(p_2\)), we lose this correspondence in the conclusion and only know that the post-states in \(q_1 \lor q_2\) are reached from the pre-states in \(p_1 \lor p_2\). As such, when merging the triples \(\uplus_F [p_1] C [e : q_1]\) and \(\uplus_F [p_2] C [e : q_2]\) into \(\uplus_F [p_1 \lor p_2] C [e : q_1 \lor q_2]\), Pulse additionally tracks the correspondence between \(p_1\) and \(q_1\) (resp. \(p_2\) and \(q_2\)). This is beneficial when later dropping branches: when dropping the disjuncts in the post-states (e.g. \(q_2\)), we can also drop their associated pre-states (\(p_2\)). This allows us to avoid accumulating ‘clutter’ in the pre-states and is tantamount to dropping a full triple rather than its post-states only.

We thus follow a similar approach here which allows us to unify FUA and BUA reasoning. More concretely, we introduce the notion of indexed disjunctions, \(P, Q \in \mathbb{N} \rightarrow \mathcal{P}(\text{STATE})\). Intuitively, an indexed disjunction \(P\) can be flattened into a standard disjunction as \(\bigvee_{i \in \text{dom}(P)} P(i)\). We write \([P] C [e : Q]\) as a shorthand for \(\text{dom}(P) = \text{dom}(Q) \land \forall i \in \text{dom}(P). (P(i) C [e : Q(i)])\), denoting a merged set of triples. Note that a triple \([p] C [e : q]\) can be simply lifted to \([P] C [e : Q]\), where \(\text{dom}(P) = \text{dom}(Q) = \{0\}\) with \(P(0) = p\) and \(Q(0) = q\). We can then use the \texttt{DisjTrack} rule (Fig. 2 on p. 11) to merge indexed disjuncts – note that the \(\text{dom}(P_1) \cap \text{dom}(P_2) = \emptyset\) premise can be simply satisfied by renaming the domain of \(P_2\). Observe that unlike the \texttt{Disj} rule, \texttt{DisjTrack} is not lossy and preserves the pre-post correspondence. Finally, the unified rule of consequence, \texttt{Cons} (Fig. 2), allows us to drop matching disjuncts from both the pre- and post-states, where \(P \downarrow \downarrow I\) denotes restricting the domain of \(P\) to \(I\). The unified \texttt{Cons} rule can be used for both FUA and BUA reasoning.

\textbf{Unified Triangles and Bug Catching Tools.} Note that the rules in Fig. 2, excluding \texttt{ConsB}, \texttt{ConsF} and \texttt{Disj} (and instead including \texttt{Cons} and \texttt{DisjTrack}) correspond to the reasoning principles used in the industrially deployed Pulse tool. That is, although Pulse is formally underpinned by IL (with FUA triples), it does not use \texttt{ConsF} and \texttt{Disj}, and instead uses \texttt{Cons} and \texttt{DisjTrack}, meaning that using our unified rules (suitable for both FUA and BUA reasoning) has no practical ramifications, and we can use Pulse as it is! This is indeed great news: in order to reason about divergence, we can extend Pulse without changing its underlying principles, and simply add our divergence rules.

\textbf{Theoretical Connection between BUA and FUA Triples.} As mentioned above, with the exception of their associated rules of consequence (\texttt{ConsF} and \texttt{ConsB} in Fig. 2) all other FUA and BUA reasoning principles and proof rules coincide. In §6 we further bolster this intuition by showing that given any under-approximate triple \([p] C [e : q]\), if \([p] C [e : q]\) is a valid FUA triple and its pre-states \(p\) are FUA-minimal, then \([p] C [e : q]\) is also a valid BUA triple. The pre-states \(p\) are FUA-minimal if for all smaller pre-states \(p' \subset p\), the triple \([p'] C [e : q]\) is not a valid FUA triple. Intuitively, this ensures that pre-states \(p\) have not been arbitrarily weakened (grown) using \texttt{ConsF}.

Conversely, we show that given an under-approximate triple \([p] C [e : q]\), if \([p] C [e : q]\) is a valid BUA triple and its post-states \(q\) are BUA-minimal, then \([p] C [e : q]\) is also a valid FUA triple. Analogously, \(q\) is BUA-minimal if for all smaller \(q' \subset q\), the triple \([p] C [e : q']\) is not a valid BUA triple. This ensures that the post-states \(q\) have not been arbitrarily weakened using \texttt{ConsB}.

\textbf{Formal Interpretation of Divergent Triples.} As discussed above, we write a divergent triple of the form \([p] C [\infty]\) to denote that \(C\) has some divergent trace(s) (i.e. in an under-approximate fashion) starting from \(p\). The next question to answer when interpreting such triples is whether there is some divergent trace starting from every state in \(p\) or some state in \(p\). Observe that both
interpretations are under-approximate as they pertain to some rather than all traces of C. Although the latter interpretation is a weaker statement, it is nevertheless sufficient for an under-approximate divergence detection framework: to establish divergence it suffices to show some divergent trace is possible from some initial state in p. However, under this weaker interpretation, inspecting a divergent triple \([p] C [\infty]\) yields little information on how the divergence arises (which may be needed for debugging and fixing the cause of divergence): as \(p\) may contain many states, it is unclear which state(s) in \(p\) lead(s) to divergence (unless \(p\) describes a single state). On the other hand, the former, stronger interpretation provides more information for debugging and fixing the cause of divergence as it states that starting from any state in \(p\) the program has a divergent trace.

Although more useful, at first glance this stronger interpretation may seem too strong and antithetic to the spirit of under-approximation in UNTer. However, this additional strength is not accompanied by a theoretical or practical cost. In theoretical terms, rather than considering an arbitrarily large set of pre-states that contain some states that may lead to divergence, one can always shrink the pre-states to contain exactly those states that lead to divergence. More concretely, when starting from a state \(s\) executing C may diverge, one can establish \([p] C [\infty]\) by defining \(p\) as the singleton set \(\{s\}\), rather than an arbitrarily large set that contains \(s\). In practical terms, this stronger interpretation incurs no additional cost when extending existing an under-approximate tool such as Pulse with divergence proof rules. In particular, the divergence rules in Fig. 3 (p. 12) fall into one of two categories: 1) base rules, where the premises contain BUA triples only (e.g. LoopFix above or Div-Loop in Fig. 3); or 2) inductive cases, where the premises contain other divergent triples (e.g. Div-Seq1 in Fig. 3) or a combination of divergent and BUA triples (e.g. Div-Seq2 in Fig. 3). For the base cases such as LoopFix, thanks to the forward reachability of BUA triples, we already established the desired result for every pre-state. Moreover, as discussed above, the BUA and FUA reasoning principles are almost identical and can be easily unified for practical purposes. As such, extending exiting under-approximate tools with a base case under a strong interpretation incurs no additional cost. Similarly, establishing an inductive case requires establishing its premises, and since neither their BUA premises (as argued above) nor their divergent premises (by inductive hypothesis) incur an additional cost, establishing an inductive case under a strong interpretation incurs no additional cost. We therefore opt for the stronger under-approximate interpretation of divergent triples: \([p] C [\infty]\) denotes that every state in \(p\) leads to some divergent trace.

4 THE UNTer FRAMEWORK

We present the UNTer framework for detecting non-termination bugs. To present the key ideas underpinning UNTer more clearly, here we develop it as an analogue of Hoare logic/incorrectness logic (IL), in that UNTer enables global and not local (compositional) reasoning as in separation logic (SL) [18] and incorrectness separation logic (ISL) [24]. Later in §7 we develop an extension of UNTer that marries the compositionality of SL/ISL with the divergence reasoning of UNTer.

**Programming Language.** To keep our presentation concise, we employ a simple imperative programming language given by the C grammar below. Our language comprises the standard constructs of skip, assignment \((x := e)\), assume statements \((\text{assume}(B))\), scoped variable declaration \((\text{local} x \in C)\), sequential composition \((C_1; C_2)\), non-deterministic choice \((C_1 + C_2)\) and loops \((C^*)\), as well as explicit error statements \((\text{error}, \text{which can be thought of e.g. as assert(false)})\).

\[
C ::= \text{skip} \mid x := e \mid \text{assume}(B) \mid \text{local} \ x \in C \mid \text{error} \mid C_1 + C_2 \mid C_1; C_2 \mid C^*
\]

As is standard, deterministic choice and loops can be encoded using their non-deterministic counterparts and assume statements. Specifically, if \((B)\) then \(C_1\) else \(C_2\) can be encoded as \((\text{assume}(B); C_1) + (\text{assume}(\neg B); C_2))\), and while \((B)\) \(C\) can be encoded as \((\text{assume}(B); C)^*; \text{assume}(\neg B))\).
**Assertions (Sets of States).** The UNTER assertion language is given by the simple grammar below, comprising classical (first-order logic) and Boolean assertions, where $\oplus \in \{=, \neq, <, \leq, \cdots \}$. Other classical connectives can be encoded using existing ones (e.g. $\neg p \equiv p \rightarrow \text{false}$). We use $p, q$, $r$ and their variants (e.g. $p'$) as metavariabes for assertions. An assertion describes a set of states, where each state is a (variable) store in $\text{STORE} \equiv \text{VAR} \rightarrow \text{VAL}$, mapping program variables to values.

$$\text{Ast} \ni p, q, r := \text{false} \mid p \Rightarrow q \mid \exists x. p \mid e \oplus e'$$

An expression $e$ is interpreted under a variable store, written as $s(e)$; this interpretation is standard and elided here. We interpret assertions as sets of states, and thus write false for $\emptyset$, $p \Rightarrow q$ for $p \subseteq q$, and $p \lor q$ for $p \lor q$. Similarly, $e \oplus e'$ denotes sets of states (stores) in which $s(e) \oplus s(e')$ holds. As discussed in §3, we introduce the notion of indexed disjunctions, $P, Q \in \mathbb{N} \rightarrow \mathcal{P}(\text{STATE})$, as a map from numbers to assertions (disjuncts); i.e. $P \equiv \bigvee_{i \in \text{dom}(P)} P(i)$.

**UNTER Under-Approximate Proof Rules for Termination.** Recall from §3 that to reason about divergence in a piecemeal fashion, we reason about terminating sub-programs via (under-approximate) BUA triples. We present the UNTER under-approximate proof rules for terminating programs in Fig. 2. The rules denoted by $\tau_F$ are FUA and BUA rules in that they are valid when we substitute in either the forward ($\tau_F$) or backward ($\tau_B$) direction. Note that as discussed in §3, with the exception of $\text{ConsF}$ and $\text{ConsB}$ rules, all rules in Fig. 2 are valid FUA and BUA triples.

- The $\text{Skip}$, $\text{Error}$, $\text{Seq}$, $\text{SeqEr}$, $\text{Choice}$, $\text{Loop0}$, $\text{Loop}$ and $\text{Disj}$ rules are identical to those of existing FUA logics [23–25]. Specifically, executing skip and error leave the state unchanged ($\text{Skip}$ and $\text{Error}$), where the former terminates normally while the latter terminates erroneously; $\text{Disj}$ allows us to merge two triples into one in a lossy fashion (as discussed in §3); the behaviour of a branching program can be under-approximated as the behaviour of some of its branches ($\text{Choice}$); and the behaviour of a loop can be under-approximated through bounded unrolling as zero ($\text{Loop0}$) or more ($\text{Loop}$) iterations. Note that while in correctness frameworks we can over-approximate a loop behaviour via an invariant, i.e. an assertion that holds after any number of iterations (including zero), in FUA/BUA frameworks we can under-approximate a loop behaviour via a subvariant as an indexed assertion $p$, where $p(n)$ describes the state after $n$ iterations. This is captured by $\text{Loop}$-Subvariant: for an arbitrary $k$, if executing $C$ terminates normally and transforms $p(n)$ to $p(n+1)$ for all $n < k$, then $p(k)$ can be reached by executing $C^*$ (i.e. executing $C$ for $k$ iterations) from the initial states $p(0)$. The $\text{SeqEr}$ captures the short-circuiting behaviour of erroneous executions: if executing $C_1$ terminates erroneously, then executing $C_1; C_2$ also terminates erroneously. By contrast, $\text{Seq}$ captures the case where executing $C_1$ does not encounter an error: if executing $C_1$ terminates normally transforming the states in $p$ to those in $r$, and executing $C_2$ terminates as $e$ (either $\text{ok}$ or $\text{er}$) and transforms $r$ to $q$, then executing $C_1; C_2$ terminates as $e$, transforming $p$ to $q$.

- The $\text{Assign}$ rule is identical to the standard Floyd assignment rule and holds for both FUA and BUA. Observe that as noted by O’Hearn [23], the Hoare assignment rule is not sound for FUA. That is, $\tau_F [p[e/x]] x := e [\text{ok}: p]$ is not sound (e.g. let $e = 42$ and $p$ be $x = y$, then the state $s \in p$ such that $s(x) = s(y) = 17$ cannot be reached by executing $x := 42$ on any state in $p[42/x]$. By contrast, the Hoare assignment rule is sound for BUA, i.e. $\tau_B [p[e/x]] x := e [\text{ok}: p]$ is a sound BUA triple. However, this difference between BUA and FUA does not have a practical ramifications as the Floyds assignment rule (in $\text{Assign}$) is sufficient to enable automated reasoning in Pulse.

- The $\text{Assume}$, $\text{Local}$ and $\text{Constancy}$ rules are analogous to the FUA rules of [23]. Concretely, executing assume($B$) terminates normally and leaves the state unchanged, provided that $B$ holds beforehand. When executing the scoped variable declaration local $x$ in $C$, the information about $x$ is erased by existentially quantifying it in the pre- and post-states. The $\text{Constancy}$ rule is used to adapt triples in different contexts and states: if an assertion $r$ holds before executing $C$, it also holds...
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Skip</strong></td>
<td>( \vdash [p] \text{skip} [ok : p] )</td>
</tr>
<tr>
<td><strong>Assign</strong></td>
<td>( \vdash [p] \ x := e \ [ok : \exists y. \ p[y/x] \land x = e[y/x]] )</td>
</tr>
<tr>
<td><strong>Assume</strong></td>
<td>( \vdash [p \land B] \text{assume}(B) [ok : p \land B] )</td>
</tr>
<tr>
<td><strong>Error</strong></td>
<td>( \vdash [p] \text{error} \ [er : p] )</td>
</tr>
<tr>
<td><strong>SEQ</strong></td>
<td>( \vdash [p] C_1 [ok : r] ) ( \vdash [r] C_2 [e : q] ) ( \vdash [p] C_1; C_2 [e : q] )</td>
</tr>
<tr>
<td><strong>Choice</strong></td>
<td>( \vdash [p] C_i [e : q] ) for some ( i \in {1, 2} )</td>
</tr>
<tr>
<td><strong>Loop0</strong></td>
<td>( \vdash [p] C^* [ok : p] )</td>
</tr>
<tr>
<td><strong>Local</strong></td>
<td>( \vdash [\exists x. p] \text{local} x \text{ in } C [e : \exists x. q] )</td>
</tr>
<tr>
<td><strong>Subst</strong></td>
<td>( \vdash [p] C [e : q] ) ( x \notin \text{fv}(p, C, q) )</td>
</tr>
<tr>
<td><strong>Loop-Subvariant</strong></td>
<td>( \forall n &lt; k. \ \vdash [p(n)] C [ok : p(n+1)] )</td>
</tr>
<tr>
<td><strong>Disj</strong></td>
<td>( \vdash [p_1] C [e : q_1] ) ( \vdash [p_2] C [e : q_2] ) ( \vdash [p_1 \lor p_2] C [e : q_1 \lor q_2] )</td>
</tr>
<tr>
<td><strong>ConsF</strong></td>
<td>( p' \subseteq p ) ( \vdash [p'] C [e : q'] ) ( q \subseteq q' ) ( \vdash [p] C [e : q] )</td>
</tr>
<tr>
<td><strong>DisjTrack</strong></td>
<td>( \vdash [p_1] C [e : Q_1] ) ( \vdash [p_2] C [e : Q_2] ) ( \vdash [p_1 \lor p_2] C [e : Q_1 \lor Q_2] )</td>
</tr>
<tr>
<td><strong>IfTrue</strong></td>
<td>( \vdash [p \land B] C_1 [e : q] ) ( \vdash [p \land B \text{ if } (B) \text{ else } C_2 [e : q]] )</td>
</tr>
<tr>
<td><strong>IfFalse</strong></td>
<td>( \vdash [p \land \neg B] C_2 [e : q] ) ( \vdash [p \land \neg B \text{ if } (B) \text{ else } C_2 [e : q]] )</td>
</tr>
<tr>
<td><strong>WhileFalse</strong></td>
<td>( \vdash [p \land \neg B] \text{ while } (B) C [ok : p \land \neg B] )</td>
</tr>
</tbody>
</table>

Fig. 2. Under-approximate proof rules where \( \vdash \) in each rule can be instantiated as \( F \) or \( B \); the highlighted rules can be derived from other rules (see §A).

Afterwards provided that it does not refer to free variables that may have been modified by \( C \). This is captured by the \( \text{fv}(r) \cap \text{mod}(C) = \emptyset \), where \( \text{fv}(r) \) denotes the free variables of \( r \) and \( \text{mod}(C) \) denotes the variables modified by \( C \) (i.e., those on the left-hand side of assignments).

As discussed in §3, ConsF and ConsB are the FUA and BUA rules of consequence, respectively. We reconcile the two in the unified rule of consequence, Cons, by using indexed disjunctions, where \( \text{dom}(P \downarrow I) = I \) and \( \forall i \in I. (P \downarrow I)(i) = P(i) \). Finally, using indexed disjunctions in DisjTrack we can merge triples in a non-lossy fashion, preserving the pre-post correspondence.

The remaining highlighted rules can be derived from existing rules (see §A). The IfTrue (resp. IfFalse) is analogous to its non-deterministic counterpart (Choice) and simply requires that the
Div-Seq1  \[ \vdash [p] C_1 \vdash [p] C_1 ; C_2 \]

Div-Seq2  \[ \vdash_B [p] C_1 [\text{ok}: q] \vdash [q] C_2 \]

Div-Choice  \[ \vdash [p] C_1 \text{ for some } i \in \{1, 2\} \]

Div-LoopUnfold  \[ \vdash [p] C ; C^* \]

Div-Loop  \[ \vdash_B [p] C [\text{ok}: q] \quad q \subseteq p \]

Div-Subvariant  \[ \forall n \in \mathbb{N}. \quad \vdash_B [p(n)] C \quad [\text{ok}: p(n+1)] \]

Div-While  \[ \vdash_B [p \land B] C \quad [\text{ok}: q \land B] \quad q \subseteq p \]

\[ \vdash [p(0) \land B] \quad \text{while } (B) C \]

Fig. 3. The UNTER divergence rules, where the highlighted rules can be derived from other rules.

In order to show that \( C_1 ; C_2 \) has a divergent trace starting from \( p \), we can show either \( C_1 \) has a divergent trace starting from \( p \) (Div-Seq1), or \( C_1 \) terminates normally transforming the states to \( q \) and \( C_2 \) does not terminate starting from \( q \) (Div-Seq2). To show that the branching program \( C_1 + C_2 \) has a divergent trace starting from \( p \), it suffices to show that some branch \( C_1 \) has a divergent trace from \( p \), i.e. in an under-approximate fashion. The Div-Cons denotes the rule of consequence for divergence: if \( C \) has some divergent trace starting from any state in \( p' \) and \( p \subseteq p' \), then \( C \) also has some divergent trace starting from any state in \( p \).

The remaining rules capture divergence for loops. Specifically, \( \text{Div-LoopUnfold} \) allows us to establish divergence after unrolling the loop once. This can be used for showing divergence in the case of nested loops, where the inner loop diverges. Specifically, using a combination of Div-Seq1 and Div-LoopUnfold we can derive Div-LoopNest as shown across, stating that if one iteration of the loop body (e.g. a nested loop) has a divergent trace, then the loop itself also has a divergent trace.

The Div-Loop rule states that if one iteration of a loop body terminates normally and transforms the states in \( p \) to ones in \( q \) (i.e. \( \vdash_B [p] C [\text{ok}: q] \) and \( q \subseteq p \), then \( C^* \) has a divergent trace starting from \( p \). Intuitively, the forward trip in the premise, \( A \vdash [p] C [\text{ok}: q] \), allows us to construct an infinite trace of \( C^* \) from any state in \( p \) given a state in \( s_0 \), (from \( A \)) executing \( C \) on \( s_0 \) results in a state \( s_1 \) in \( q \subseteq p \), and thus (from \( A \)) executing \( C \) on \( s_1 \) results in a state \( s_2 \) in \( q \subseteq p \), ad infinitum.
The Div-Subvariant is the subvariant rule for divergence: if an iteration of the loop body terminates normally and transforms $p(n)$ to $p(n+1)$ for an arbitrary $n$, then $C^*$ has a divergent trace starting from the initial states $p(0)$. Note that given any loop body $C$, if $C$ does not contain a conditional (if or while) statement and executing $C$ does not encounter an error, then the non-deterministic loop $C^*$ always has a divergent trace. However, this is not necessarily the case with conditional if/while statements (encoded via assume statements). This is illustrated in the Div-While rule, requiring that the loop condition $B$ hold at the end of an iteration, which is not always the case. For instance, for while $(x = 0) \ x := 1$ we fail to establish $x = 0$ after an iteration of $x := 1$.

As before, all highlighted rules in Fig. 3 can be derived from other rules (see §A). For instance, Div-WhileNest can be derived from Div-LoopNest, Seq and Assume.

5 EXAMPLES

We present several simple examples of divergent programs (with divergent loops) and demonstrate how we can use our UNTer proof system to detect them. All divergent behaviours presented here, and many more, have also been detected using our Pulse prototype (see §8).

Example 1 (Fig. 4a). Consider the simple example in Fig. 4a comprising a simple divergent loop. We can detect this using Div-While (with $p = q = true$) as shown below:

$$
\begin{align*}
\vdash_B [x = 0] & \text{skip} \ [ok : x = 0] \quad \text{(Skip)} \\
& \quad \vdash_B [x = 0] \text{while } (x = 0) \text{skip} \ [\infty] \quad \text{(Div-While)}
\end{align*}
$$

Example 2 (Fig. 4b). Consider the simple example in Fig. 4b comprising a simple while loop with a buggy check. We can detect this using Div-While (with $p = true$ and $q = x > 1$) as shown below:

$$
\begin{align*}
\vdash_B [x \geq 0 \ x := x+1 \ [ok : \exists v. v \geq 0 \land x = v+1] \quad \text{(Assign)} \\
\vdash_B [x \geq 0 \ x := x+1 \ [ok : x \geq 1 \land x \geq 0] \quad \text{(ConsEq)} \\
& \quad \vdash_B [x \geq 0] \text{while } (x \geq 0) \ x := x+1 \ [\infty] \quad \text{(Div-While)}
\end{align*}
$$

Example 3 (Fig. 4c). Consider the example in Fig. 4c. Prior to the first iteration of the loop $x+y = 3$ holds, and although the values of $x$ and $y$ are updated in each iteration, their sum remains unchanged after each iteration (i.e. $x+y = 3$) and thus the loop diverges. We present an UNTer proof outline of this divergent behaviour on the left of Fig. 5. For brevity, rather than giving full derivations, we follow the classical Hoare logic proof outline, annotating each line of the code with its pre- and post-states. We further commentate each proof step and write e.g. // Assign to denote an application of Assign. As in Hoare logic proof outlines, we assume that Seq is applied at every step; i.e. later instructions are executed only if the earlier ones execute normally (with ok).
Let $p \triangleq x+y = 3 \land x+y > 1$; after the initial assignment to $x$ and $y$ and applications of ConsEq and Div-Cons, we establish $p$ (line 6). We then apply Div-While (lines 6–14) to show that the loop body leaves the set of states $p$ unchanged (lines 8–13). The proof of lines 8–13 is then straightforward, and simply involves the applications of Assign and ConsEq.

**Example 4 (Fig. 4d).** Consider the example in Fig. 4d. At first glance it may seem that the loop terminates since the value of $y$ is incremented in the else branch of each iteration. However, starting from $y = 0$, the then branch is taken in each iteration (since $y \leq 50$) and thus $y$ is never incremented, resulting in divergence. We present an UNTER proof outline of this divergent behaviour on the right of Fig. 5. After applying ConsEq to rewrite $p$ equivalently as $p \land y \leq 50$ (line 5), we apply InTrue to show we can take the then branch and arrive at $p$ (lines 7–10).

**Example 5 (Fig. 4e).** Consider the example in Fig. 4e with nested loops. Note that the value of $x$ is incremented at the end of each iteration of the inner loop and thus the inner loop terminates. By contrast, although $y$ is incremented at the end of each iteration of the outer loop and thus it may seem at first glance that the outer loop terminates, on closer inspection the value of $y$ us reset to 0 in the last iteration of the inner loop. As such, at the end of each iteration of the outer loop $y$ is incremented and updated 1, and thus the outer loop diverges.

We present an UNTER proof outline of this at the top of Fig. 6. After applying Div-Cons to obtain $y < 100$, we apply Div-While (lines 2–23) to show that the loop body leaves $y < 100$ unchanged (lines 4–22). After the assignment on line 5, we apply ConsEq to rewrite the states as $p(0) \land x \leq 100$ (line 7), with $p(n)$ defined below the proof at the top of Fig. 6. We then apply WhileSubvariant to show that at the end of the execution of the inner loop we arrive at $y = 0 \land x = 101 \land x \leq 100$ (lines 7–21). Note that WhileSubvariant has two premises, which we establish in two columns on lines 9–14 and 15–20. On lines 9–14 we show that for $n < 100$, each iteration of the loop transforms $p(n) \land x < 100$ to $p(n+1) \land x < 100$; on lines 15–20 we show that in the final iteration of the loop with $p(100)$ (i.e. when $x = 100$), we reset $y$ to 0 and increment $x$, arriving at $y = 0 \land x = 101 \land x < 100$ which is included in $y < 100$ (line 22), as per the second premise of Div-While.
1. \( [x = 0 \land y = 0] \) // Div-Cons
2. \( [y < 100] \)
3. while \( (y < 100) \)
   4. \( [y < 100] \) // Assign
   5. \( x := 0 \) // Assign
   6. \( [ok: y < 100 \land x = 0] \) // ConsEq
   7. \( [ok: p(0) \land x \leq 100] \)
   8. while \( (x \leq 100) \)
   9. \( \forall n < 100. [p(n) \land n < 100 \land x \leq 100] \)
   10. if \( (x = 100) \) \( y := 0 \) // IfFalse, Skip
   11. else skip // IfTrue, Assign
   12. \( [ok: p(n) \land n < 100 \land x \leq 100] \)
   13. \( x := x + 1 \) // Assign, ConsEq
   14. \( [ok: p(n + 1) \land x \leq 100] \)
   21. \( [ok: y = 0 \land x = 101 \land x \neq 100] \)
   22. \( [ok: y < 100] \)

where for all \( n \in \mathbb{N} \) : \( p(n) \triangleq x = n \land y < 100 \)

1. \( [x = 0 \land y = 0] \)
2. \( x := 42; y := 1 \) // Assign, ConsEq
3. \( [ok: x = 42 \land y = 1] \) // Div-Cons
4. \( [ok: x \leq 100 \land y < 100] \)
5. while \( (y < 100) \)
   6. \( [x \leq 100 \land y < 100] \) // Div-Cons
   7. \( [x \leq 100] \)
   8. while \( (x \leq 100) \)
   9. \( [x \leq 100] \) // ConsEq
   10. \( [x < 100 \lor x = 100] \)
   11. \( [x < 100] \)
   12. if \( (x = 100) \) \( x := 1; y := 2 \times y \)
   13. else \( x := x + 1 \)
   14. \( [ok: x \leq 100] \)
   15. \( [x = 100] \)
   16. if \( (x = 100) \) \( x := 1; y := 2 \times y \)
   17. else \( x := x + 1 \)
   18. \( [ok: x \leq 100] \)
   20. \( [\infty] \)

Fig. 6. Proof sketch of divergence in Fig. 4e (above), where the two columns on lines 9–14 and 15–20 denote the proof sketches of the two premises of WhileSubvariant; proof sketch of divergence in Fig. 4f (below), where the two columns on lines 11–14 and 15–18 denote the proof sketches of the two premises of Disj.

**Example 6 (Fig. 4f).** Consider the nested loops in Fig. 4f. Note that starting with \( x = 42 \) (after the initial assignment), the else branch of the inner loop increments \( x \) in all but the last iteration of the inner loop (since \( x = 100 \)), whereupon the value of \( x \) is reset to 1; i.e. the inner loop diverges.

We present an UNTer proof outline of this divergent behaviour at the bottom of Fig. 6. After the initial assignments (line 2) and applying Div-Cons to arrive at \( x \leq 100 \land y < 100 \) (line 4), we apply Div-WhileNest (lines 4–21) to show that the loop body diverges (lines 6–20). Once again, we apply
\textbf{6 THE UNTer MODEL AND SEMANTICS}

\textbf{UNTer Operational Semantics.} Although in sequential settings the semantics is typically given in the big-step fashion \cite{23, 24}, we opt for \textit{small-step} semantics instead. This is because big-step semantics by definition describe \textit{terminating} executions, while our aim is to formalise \textit{divergent} triples. Specifically, we formalise the semantics of a divergent triple as an \textit{infinite}, non-terminating execution trace. The UNTer small-step transitions are straightforward and are of the form $C, s \rightarrow C', s', \epsilon$, where $C$ and $s$ respectively denote the current command and store (state), $C'$ and $s'$ denote their continuations (what they reduce to) and $\epsilon$ denotes the exit condition, describing whether reducing $C$ to $C'$ took place normally (\textit{ok}) or erroneously (\textit{er}). For brevity we present the UNTer small-step transitions in the technical appendix (§B.1).

\textbf{Semantic BUA and FUA Triples.} Recall that intuitively a BUA triple $\vdash_B [p] C [\epsilon : q]$ states that every pre-state $s_p$ in $p$ reaches some post-state $s_q$ in $q$ under $\epsilon$ by executing $C$. Analogously, a FUA triple $\vdash_F [p] C [\epsilon : q]$ states that every post-state $s_q$ in $q$ can be reached from some pre-state $s_p$ in $p$ under $\epsilon$ by executing $C$. Put formally, in both cases we must have $C, s_p \overset{n}{\rightarrow} s_q, \epsilon$, denoting that executing $C$ \textit{terminates} after $n$ steps under $\epsilon$ and transforms $s_p$ to $s_q$ (see Def. 1 below).

\textbf{Definition 1} (Semantic BUA and FUA triples). A BUA triple is \textit{valid}, written $\models_B [p] C [\epsilon : q]$, iff for all $s_p \in p$, there exists $s_q \in q$ and $n$ such that $C, s_p \overset{n}{\rightarrow} s_q, \epsilon$, where:

$$C, s \overset{n}{\rightarrow} C', s', \epsilon \iff (n=0 \land C=C'=\text{skip} \land s=s' \land \epsilon=\text{ok}) \lor (n=1 \land \epsilon \in \text{ErExit} \land C, s \rightarrow C', s', \epsilon)$$

$$\lor (\exists k, C'', s'', n=k+1 \land C, s \rightarrow C'', s'', \epsilon \lor C'' \rightarrow C', s', \epsilon)$$

and $C, s \rightarrow C', s', \epsilon$ is the UNTer small-step transitions given in §B.1 (Fig. 8). A FUA triple is \textit{valid}, written $\models_F [p] C [\epsilon : q]$, iff for all $s_q \in q$, there exists $s_p \in p$ and $n$ such that $C, s_p \overset{n}{\rightarrow} s_q, \epsilon$.

The first disjunct in $C, s \overset{n}{\rightarrow} C', s', \epsilon$ states that any state can be reached under \textit{ok} in zero steps without changing the underlying state, provided that $C$ is simply skip. The second disjunct captures the short-circuit semantics of errors: a state $s'$ can be reached in one step under \textit{er} when $C$ takes an erroneous step. Analogously, the last disjunct captures the inductive cases ($n=k+1$), where $C$ takes an \textit{ok} step, and $s'$ is subsequently reached in $k$ steps under $\epsilon$.

We next show that the BUA and FUA proof systems presented in Fig. 2 are both \textit{sound} and \textit{complete}, with the full proof given in the technical appendix (§B.2 and §C.1).

\textbf{Theorem 7} (BUA and FUA soundness). For all $p, q, C$ and $\epsilon$:

1) if $\vdash_B [p] C [\epsilon : q]$ is derivable using the rules in Fig. 2, then $\models_B [p] C [\epsilon : q]$ holds; and

2) if $\vdash_F [p] C [\epsilon : q]$ is derivable using the rules in Fig. 2, then $\models_F [p] C [\epsilon : q]$ holds.

\textbf{Theorem 8} (BUA and FUA completeness). For all $p, q, C$ and $\epsilon$:

1) if $\models_B [p] C [\epsilon : q]$ holds, then $\vdash_B [p] C [\epsilon : q]$ is derivable using the rules in Fig. 2; and

2) if $\models_F [p] C [\epsilon : q]$ holds, then $\vdash_F [p] C [\epsilon : q]$ is derivable using the rules in Fig. 2.

We next present the formal interpretation of divergent triples.
**Definition 2** (Semantic divergent triples). A divergent triple is valid, written $\models [p] C [\infty]$, iff for all $s \in p$, there exists an infinite series of $C_1, C_2, \ldots$ and $n_1, n_2, \ldots$ such that $C, s \sim_n C_1, s_1, ok \sim_{n_2} C_2, s_2, ok \sim_{n_3} \cdots$, where the chain $C, s \sim_{n_1} C_1, s_1, ok \sim_{n_2} C_2, s_2, ok \sim_{n_3} \cdots$ is a shorthand for $C, s \sim_{n_1} C_1, s_1, ok \land C_1, s_1 \sim_{n_2} C_2, s_2, ok \land \cdots$, and $\sim_n$ is defined as follows:

$$C, s \sim_n C', s', \epsilon \iff \ \begin{align*}
    & (n = 1 \land C, s \rightarrow C', s', \epsilon) \\
    & \lor \ (\exists k, s'', C''. \ n = k+1 \land C, s \rightarrow C'', s'', ok \land C'', s'' \sim^k C', s', \epsilon)
\end{align*}$$

Note that unlike the $C, s \rightarrow C', s'$ transitions in Def. 1 which describe terminating traces (either via short-circuiting or by reduction to skip), the $C, s \sim_n C', s'$ transitions do not stipulate termination and simply state that executing $C$ from $s$ for $n$ steps reduces to $C'$ and results in $s'$.

We next formalise the relationship between FUA and BUA triples (see p. 8), with the proof in §D.

**Theorem 9.** For all $p, C, q, \epsilon$:
1) if $\models_F [p] C \mid \epsilon \rightarrow q$ and $\text{min}_p (p, C, q)$ hold, then $\models_B [p] C \mid \epsilon \rightarrow q$ also holds; and
2) if $\models_B [p] C \mid \epsilon \rightarrow q$ and $\text{min}_p (p, C, q)$ hold, then $\models_F [p] C \mid \epsilon \rightarrow q$ also holds, where:

$$\text{min}_p (p, C, q) \iff \forall p', p' \subset p \Rightarrow \models_F [p'] C \mid \epsilon \rightarrow q$$

$$\text{min}_p (p, C, q) \iff \forall q', q' \subset q \Rightarrow \models_B [p] C \mid \epsilon \rightarrow q'$$

Finally, we show that the divergence proof system presented in Fig. 3 is both sound and complete, with the full proof given in the technical appendix (§B.3 and §C.2).

**Theorem 10** (Divergence soundness and completeness). For all $p$ and $C$, if $\vdash [p] C [\infty]$ is derivable using the rules in Fig. 3, then $\models [p] C [\infty]$ holds. For all $p$ and $C$, if $\models [p] C [\infty]$ holds, then $\vdash [p] C [\infty]$ is derivable using the rules in Fig. 3.

### 7 EXTENSION TO SEPARATION LOGIC

We describe how we develop UNTERSL by extending UNTER with the compositional reasoning principles of separation logic (SL) [18]. Raad et al. [24] have developed incorrectness separation logic (ISL) by extending the FUA-based incorrectness logic (IL) [23] with separation logic. As Raad et al. [24] argue, the original model of SL is unsound for FUA reasoning, and thus they adapt the original model to recover the soundness of ISL (see §E for details). We adopt the model of Raad et al. [24] and show that it is also sound for BUA reasoning.

**UNTERSL Programming Language and Assertions.** To account for operations that access the heap, in UNTERSL we extend our programming language from §4 with the following heap-manipulating operations (below, left) for allocation ($x := \text{alloc}()$), deallocation ($\text{free}(x)$), reading from the heap ($\text{lookup}, x := [y]$) and writing to the heap (mutation, $[x] := y$). We similarly extend the UNTER assertions as follows (below, right) by adding structural assertions to describe heaps.

```
\text{COMM} ::= \cdots \mid x := \text{alloc}() \mid \text{free}(x)
\text{AST} ::= \cdots \mid \text{emp} \mid e \mapsto e'

|x := [y] \mid [x] := y \mid e \mapsto e' \mid e \not\mapsto \mid p \not\mapsto q
```

The UNTERSL assertions describe sets of states, where a state comprises a (variable) store and a heap. The existing UNTER assertions from §4 then simply describe states in which the heap is empty and the store satisfies the assertion (as in UNTER). The structural assertions above are those of ISL [24] (which themselves are those of SL [18] extended with $e \not\mapsto$), and describe a set of states by constraining the shape of the underlying heap. More concretely, emp describes states in which the heap is empty; $e \mapsto e'$ describes states in which the heap comprises a single location denoted by $e$ containing the value denoted by $e'$; similarly, $e \not\mapsto$ describes states in which the heap comprises a single location at $e$ containing the designated value $\bot$; and $p \not\mapsto q$ describes states in which the heap can be split into two disjoint sub-heaps, one satisfying $p$ and the other $q$. Note that whilst $e \mapsto e'$...
Fig. 7. **UNTerSL** proof rules (excerpt), where $x$ and $x'$ are distinct variables and $\vdash$ in each rule can be instantiated as $F$ or $B$; see Fig. 9 in §E for the full set of **UNTerSL** rules.

states that the location at $e$ is allocated (and contains value $e'$), $e \nrightarrow$ states that the location at $e$ is deallocated. We write $e \rightarrow$ as a shorthand for $\exists v. e \rightarrow v$.

**UNTerSL Proof Rules (Syntactic UNTerSL Triples).** We present an excerpt of the **UNTerSL** proof rules in Fig. 7; please see Fig. 9 in §E for the full set of rules. Note that all **UNTerSL** rules (both BUA and FUA) in Fig. 2, except **Constancy** and **Assign**, are also **UNTerSL** rules and are omitted from Fig. 9. In particular, we replace **Constancy** with the more powerful **Frame** rule and give a local rule for assignment (see below). As with ISL (and in contrast to **UNTerSL**), **UNTerSL** triples are local in that their pre-states only contain the resources needed by the program. For instance, as assignment requires no heap resources, as shown in **AssignSL** the pre-state of skip is simply given by the pure (non-heap) assertion $x = x'$, recording the old value of $x$ which can be used in the post-state.

As in SL and ISL, the crux of **UNTerSL** lies in the **Frame** rule, allowing one to extend the pre- and post-states with disjoint resources in $r$, where $\text{fv}(r)$ returns the set of free variables in $r$, and $\text{mod}(C)$ returns the set of (program) variables modified by $C$ (i.e. those on the left-hand of $\colon \Rightarrow$ in assignment, lookup and allocation). These definitions are standard and elided. Heap manipulation rule are identical to those of ISL. For instance, **Store** describes a successful heap mutation, while **StoreEr** and **StoreNull** state that mutating $x$ causes an error when $x$ is deallocated or null, respectively.

The **UNTerSL** divergent proof rules are identical to those of **UNTer** in Fig. 3, except that the terminating (BUA) **UNTerSL** triples in the premises (e.g. the first premise of **Div-Seq2**) are replaced with their **UNTerSL** counterparts. Additionally, we can extend the framing principle to divergent triples as shown in **Div-Frame**. That is, if $C$ has a divergent trace starting from the states in $p$, then it also has divergent traces starting from the states in $p * r$.

**UNTerSL Model and Semantics.** As well as a (variable) store, in **UNTerSL** each state additionally includes a heap (memory); i.e. an **UNTerSL** state, $\sigma \in \text{STATE}_{SL} \equiv \text{STORE} \times \text{HEAP}$, is a pair of the form $(s, h)$, comprising a store $s \in \text{STORE} \equiv \text{VAR} \rightarrow \text{VAL}$ (as in **UNTer**) and a heap $h \in \text{HEAP}$. The set of heaps is $\text{HEAP} \equiv \text{LOC} \rightarrow \text{VAL} \cup \{\bot\}$; that is, each heap is a partial map from locations to either values (for allocated locations) or the designated $\bot$ value (for deallocated locations).

The semantics of **UNTerSL** assertions are as those of ISL and elided here (see §E). As with **UNTer**, we define the **UNTerSL** semantics through small-step transitions, where the semantics of constructs imported from **UNTer** are as in **UNTer** and are simply lifted to operate on **UNTerSL** states. The transitions of to heap-manipulating operations are standard and elided here (see Fig. 10 in §E).

**Semantic BUA, FUA and Divergent triples in UNTerSL.** The formal interpretations of BUA, FUA and divergent triples in **UNTerSL** are identical to their **UNTer** counterparts, except that the **UNTer** states (stores) are replaced with corresponding **UNTerSL** states (pairs of stores and heaps).

More concretely, a BUA triple in **UNTerSL** is valid, written $\models_B [p] C [e : q]$, iff for all $\sigma_p \in p$, there exists $\sigma_q \in q$ and $n$ such that $C, \sigma_p \xrightarrow{n} \sigma_q, e$, where $C, \sigma \xrightarrow{n} \sigma', e$ is as defined in Def. 1 with the **UNTer** states $s, s', s''$ replaced with corresponding **UNTerSL** states $\sigma, \sigma'$ and $\sigma''$, and where
We describe our work-in-progress prototype implementation, \( \text{Pulse}^{\infty} \), based on \( \text{UNTer}^{\infty} \) theory and as an extension of Pulse. \( \text{Pulse}^{\infty} \) currently only detects the most obvious kinds of divergence bugs than can be characterised by \( \text{UNTer}^{\infty} \). As such, we plan on adding more features to \( \text{Pulse}^{\infty} \) to support detection of additional divergence bug classes, including function calls, gotos and exception handling, which are all control-flow patterns that are supported by Pulse out of the box.

\( \text{Pulse}^{\infty} \) Execution Domain. We generalise the Pulse execution domain in \( \text{Pulse}^{\infty} \) by adding a new kind of error state \( \text{InfiniteExecution} \) on top of the existing \( \text{ok} \) and \( \text{er} \) states of Pulse. For every back-edge of the program, \( \text{Pulse}^{\infty} \) checks the lasso property between the pre- and the post-states as \( [p] \) \( C^* \) \( \text{ok} : p \); i.e. there exists a pre-state before the back-edge that is also a post-state. Each Pulse state contains 1) a disjunctive part that encodes the set of reachable states in a big disjunction, one disjunct per path, without merging path conditions; and 2) a non-disjunctive part that encodes other environment conditions that hold for all paths. \( \text{Pulse}^{\infty} \) retains this product domain structure and our divergence extension only requires updating the disjunctive part of the Pulse state.

Abstract Interpreter. The Pulse checker implementation is based on an abstract interpretation subsystem of Infer known as \( \text{Infer.AI} \), providing generic abstract domain primitives (e.g. top, bottom, and join) as well as generic widening and narrowing extensions for convergence acceleration. \( \text{Pulse}^{\infty} \), as with Pulse, only uses widening to encode visiting the analysed program back-edges. Unlike Pulse, however, in \( \text{Pulse}^{\infty} \) we also need to define widening for the disjunctive domain part of the state to check that a given state \( \sigma_p \) is reachable from itself for a given path. If such condition is found during widening, the new \( \text{InfiniteExecution} \) error state is added to the post-condition, and this error state is eventually reported when it bubbles up in the active Pulse state queue.

Scalability. The abduction and separation logic features of Pulse allow our analysis to be scalable, and running \( \text{Pulse}^{\infty} \) on thousands of projects yields no perceptible performance change compared to Pulse, thus validating Pulse as a potential framework for compositional non-termination proving in practice. Further development and evaluation of \( \text{Pulse}^{\infty} \) at scale is planned for future work.

9 RELATED WORK

There are of course very many individual reports of personal experience with non-termination bugs which many readers will no doubt have experienced. Our work gathering CVE’s related to non-termination was an attempt to collect data on important such bugs occurring in practice. A
recent empirical study is also worth noting, which looked at non-termination bugs in OSS projects, finding 445 non-termination bugs from 3,142 GitHub commits [27].

There has been significant work on automated methods for proving termination; see the survey by Cook et al. [11]. When a termination prover fails, the question of whether the failed proof identifies a termination bug or if it is a false positive is more difficult than proving safety: termination bugs cannot be generally witnessed with finite traces (assuming unbounded resources in the computation model, that is). However, as Godefroid argues [16], the main value of analysis tools lies in the discovery of bugs, not in the proof of program correctness. Thus, it is valuable to consider proving non-termination, even without waiting for the wide deployment of termination verifiers.

The fundamental work of Gupta et al. [17] looked at using proof to find non-termination bugs. They work with a transition system consisting of initial and final states and a transition relation, and they identify the concept of a recurrence set $R$ as (i) a non-empty intersection with the initial set of states, and (ii) reachability of $R$ from every state satisfying $R$. Reachability in (ii) corresponds to $\forall_B \left[ R \right] C \left[ ok : R \right]$. One might argue that the relation between the UNTER proof system for $\forall_B \left[ p \right] C \left[ ok : q \right]$ and the model of Gupta et al. [17] is analogous to the relation between Hoare’s logic and Floyd’s proof method [1]: using the under-approximate triples provides a route to compositional reasoning. There are many detailed differences beyond these points. They first run a concolic executor to gather assertions at program points, especially loop entry, but then employ an encoding in arithmetic to determine reachability facts for loop bodies, and they treat the heap concretely (as this encoding is difficult otherwise). By contrast, we reason about reachability both of the loop stems and bodies in the same logical system, and we use separation logic to reason abstractly about heaps (SL-based analyses were not available at the time of Gupta et al. [17]).

Our prototype, Pulse\textsuperscript{∞}, inherits the strengths and weaknesses of Pulse. In terms of its strengths, it is easy to run Pulse\textsuperscript{∞} on program snippets, to scale it to large programs, and to incorporate it in a CI-based deployment on pull requests. In terms of its weaknesses, Pulse has a weak treatment of arithmetic, meaning that tricky examples (as in [17]) may not be provable. The strengths and weaknesses of [17] are the converse. We do not believe the weaknesses of either are inevitable; e.g. by adding a stronger arithmetic solver to Pulse\textsuperscript{∞} it would obviously be possible to prove tricky examples; the question is the effect this would have on performance.

After Gupta et al. [17], there have been many further papers on automatic non-termination proving or checking. Cook et al. [8], Chen et al. [7] introduce novel ideas on the use of over-approximation, going beyond the under-approximate logics here. Le et al. [20] introduce a separation logic for proving both termination and non-termination, using temporal predicates in preconditions, and we are not sure of the relation to the under-approximate approach here.

The idea of finding non-termination bugs using proof is appealing, and it is perhaps not intuitively too complicated. Although this paper is but a step on the way, it is not unreasonable to hope that non-termination proof techniques, with further maturation, might be developed to a degree where they could be routinely deployed in engineering practice.

REFERENCES


A DERIVED RULES

IFTRUE Derivation

\[ \vdash [p \land B] \quad \text{(ASSUME)} \]

\[ \vdash [p \land B] \quad \text{(given)} \]

\[ \vdash [p \land B] \quad \text{(SEQ)} \]

\[ \vdash [p \land B] \quad \text{if } C_1 \text{ else } C_2 \quad \text{(CHOICE)} \]

\[ \vdash [p \land B] \quad \text{if } B \text{ then } C_1 \text{ else } C_2 \quad \text{(If encoding)} \]

IFFALSE Derivation

\[ \vdash [p \land \neg B] \quad \text{(ASSUME)} \]

\[ \vdash [p \land \neg B] \quad \text{(given)} \]

\[ \vdash [p \land \neg B] \quad \text{(SEQ)} \]

\[ \vdash [p \land \neg B] \quad \text{if } C_1 \text{ else } C_2 \quad \text{(CHOICE)} \]

\[ \vdash [p \land \neg B] \quad \text{if } C_1 \text{ else } C_2 \quad \text{(If encoding)} \]

CONS_EQ Derivation (BUA case)

\[ p \equiv p' \quad \text{(given)} \]

\[ p \subseteq p' \quad \vdash_B [p'] \quad \text{(given)} \]

\[ q \equiv q' \quad \text{(given)} \]

\[ q' \subseteq q \quad \vdash_B [p'] \quad \text{(CONS)} \]

CONS_EQ Derivation (FUA case)

\[ p \equiv p' \quad \text{(given)} \]

\[ p' \subseteq p \quad \vdash_F [p'] \quad \text{(given)} \]

\[ q \equiv q' \quad \text{(given)} \]

\[ q \subseteq q' \quad \vdash_F [p'] \quad \text{(CONS)} \]

WHILE_FALSE Derivation

\[ \vdash [p \land \neg B] \quad \text{(ASSUME)} \]

\[ \vdash [p \land \neg B] \quad \text{(SEQ)} \]

\[ \vdash [p \land \neg B] \quad \text{while } (B) \quad \text{(while encoding)} \]

WHILE_SUBVARIANT Derivation

In the following, let \( r(n) \equiv p(n) \land B \) for all \( n \in \mathbb{N} \):

\[ (1) \quad \vdash [p(0) \land B] \quad \text{(SEQ)} \]

\[ \vdash [p(0) \land B] \quad \text{(LOOP)} \]

\[ \vdash [q \land \neg B] \quad \text{(SEQ)} \]

\[ \vdash [q \land \neg B] \quad \text{while } (B) \quad \text{(while encoding)} \]
∀n < k. \( \vdash r(n) \) assume\( (B) \) \([ ok: r(n) ]\)

\oppelk\( r(n) \) assume\( (B) \) \([ ok: r(n+1) ]\)

\( \vdash r(0) \) (assume\( (B); C) [ ok: r(k) ]\)

\( \vdash [ p(0) \land B ] \) (assume\( (B); C) [ ok: p(k) \land B ]\)

(definition of \( r \))

(1)

\( \vdash [ p(k) \land B ] \) assume\( (B) \) \([ ok: p(k) \land B ]\)

\( \vdash [ p(k) \land B ] \) C [ ok: \( q \land \neg B \)]

(given)

(SEQ)

(2)

\( \text{Div-LoopNest Derivation} \)

In the following, let \( q(n) \equiv p(n) \land B \) for all \( n \in \mathbb{N} \):

\( \frac{}{[ p ] \ C \ [ \infty ]} \) (given)

(Div-Seq1)

\( \frac{}{[ p ] \ C; C^* \ [ \infty ]} \) (Div-LoopUnfold)

\( \text{Div-While Derivation} \)

\( \vdash_B [ p \land B ] \) assume\( (B) \) \([ ok: p \land B ]\)

\( \vdash_B [ p \land B ] \) C [ ok: \( q \land B \)]

(given)

(SEQ)

\( q \subseteq p \) (given)

(Div-Loop)

\( q \land B \subseteq p \land B \) (Div-Seq1)

(while encoding)

\( \text{Div-WhileNest Derivation} \)

\( \vdash_B [ p \land B ] \) assume\( (B) \) \([ ok: p \land B ]\)

\( \vdash_B [ p \land B ] \) C [ \infty ]

(given)

(Div-Seq2)

\( \vdash_B [ p \land B ] \) (assume\( (B); C) [ \infty ]\)

(Div-LoopNest)

\( [ p \land B ] \) (assume\( (B); C) [ \infty ]\)

(Div-Seq1)

(while encoding)

\( \text{Div-WhileSubvariant Derivation} \)

In the following, let \( q(n) \equiv p(n) \land B \) for all \( n \in \mathbb{N} \):
∀ 𝑛 ∈ ℎ. ⊢ B[𝑞(𝑛)] (ASSUME) ∀ 𝑛 ∈ ℎ. ⊢ B[𝑞(𝑛)+1] (given)

∀ 𝑛 ∈ ℎ. ⊢ B[𝑞(𝑛)] (ASSUME) ∀ 𝑛 ∈ ℎ. ⊢ B[𝑞(𝑛)+1] (SEQ)

∀ 𝑛 ∈ ℎ. ⊢ B[𝑞(𝑛)] (DIV-SUBVARIANT)

[𝑞(0)] (assume(B); C)* [∞] (definition of 𝑞(0))

[𝑝(0) ∧ 𝑄] (assume(B); C)* [∞] (DIV-SEQ1)

[𝑝(0) ∧ 𝑄] while (𝐵) [∞] (while encoding)
S-Local
\[ s' = s[x \mapsto v] \quad v \in \text{Val} \]
\[ \text{local } x \text{ in } C, s \rightarrow C; \text{end}(x,s(x)), s' \]

S-LocalEnd
\[ s' = s[x \mapsto v] \]
\[ \text{end}(x,v), s \rightarrow \text{skip}, s' \]

S-Assign
\[ s' = s[x \mapsto s(e)] \]
\[ x := e, s \rightarrow \text{skip}, s', \text{ok} \]

S-Assume
\[ s(B) = \text{true} \]
\[ \text{assume}(B), s \rightarrow \text{skip}, s, \text{ok} \]

S-Error
\[ \text{error}, s \rightarrow \text{skip}, s, e \]

S-Choice
\[ i \in \{1,2\} \]
\[ C_1 + C_2, s \rightarrow C_i, s, \text{ok} \]

S-Seq1
\[ C_1; s \rightarrow C_1', s', e \]
\[ C_1; C_2, s \rightarrow C_1'; C_2', s', e \]

S-SeqSkip
\[ \text{skip}; C, s \rightarrow C, s, \text{ok} \]

S-Loop
\[ C^*, s \rightarrow \text{skip}, s, \text{ok} \]
\[ C^*, s \rightarrow C; C^*, s, \text{ok} \]

Fig. 8. The UNTer small-step operational semantics

B UNTER SEMANTICS AND SOUNDNESS

Instrumented Commands and Operational Semantics. Although in sequential settings the semantics is given in the big-step fashion [23, 24], we opt for small-step semantics instead. This is because big-step semantics by definition describe terminating executions, while our aim is to formalise the semantics of divergent triples. Specifically, as we describe below, we formalise the semantics of a divergent triple as an infinite, non-terminating execution trace.

Note that local \( x \) in \( C \) declares a variable \( x \) whose scope is limited to \( C \). To describe the semantics of local \( x \) in \( C \) in a small-step fashion, we introduce instrumented commands, defined by the grammar below (where \( C \) is as defined in §4), which additionally include the \( \text{end}(x,v) \) construct, recording the existing (old) value of \( x \) when redeclaring \( x \) in a new scope.

\[ C := C \mid \text{end}(x,v) \mid C_1; C_2 \]

We present our small-step semantics in Fig. 8, with transitions of the form \( C, s \rightarrow C', s', e \), where \( C \) and \( s \) respectively denote the current (instrumented) command and store (state), \( C' \) and \( s' \) denote their continuations (what they reduce to) and \( e \) denotes the exit condition, describing whether reducing \( C \) to \( C' \) took place normally (\( \text{ok} \)) or erroneously (\( \text{er} \)). As shown in S-Local, when evaluating local \( x \) in \( C \) under a state \( s \in \text{STORE} \), we assign an arbitrary value \( v \) to \( x \) in \( s \), and continue with executing \( C \) followed by \( \text{end}(x,s(x)) \). That is, we record the existing value of \( x, s(x) \), so that we can restore it once the execution of \( C \) has ended, as reflected in the S-LocalEnd transition.

The remaining transition rules are standard: assigning \( e \) to \( x \) simply evaluates \( e \) in the current state (denoted by \( s(e) \)) and updates the value of \( x \) in the state, terminating normally; assume(\( B \)) reduces to skip normally when \( B \) evaluates to true in the current state; error reduces to skip erroneously; and \( C_1 + C_2 \) non-deterministically reduces to one of its branches (\( C_i \) with \( i \in \{1,2\} \)). When reducing \( C_1; C_2 \), we either reduce the left-hand side until it reduces to skip (S-Seq1), or continue with the right-hand side when the left side is skip (S-SeqSkip). Finally, we either reduce a loop to skip, i.e. unroll it zero times (S-Loop0), or unroll it once and continue with \( C^* \) (S-Loop).

B.1 UNTER Semantics

Lemma 1. For all \( n, s, s', C, C' \), if \( C, s \rightarrow_n C', s', \text{ok} \), then \( C' = \text{skip} \).

Proof. By induction on \( n \).

Base case \( n=0 \)

Pick arbitrary \( s, s', C, C' \) such that \( C, s \rightarrow_0 C', s', \text{ok} \). From the definition of \( \rightarrow_0 \) we then have \( C' = \text{skip} \), as required.
**Inductive case** $n=k+1$

Pick arbitrary $s, s', C, C'$ such that $C, s \xrightarrow{n} C', s', ok$. From the definition of $\xrightarrow{n}$ we know there exists $C'', s''$ such that $C, s \xrightarrow{i} C'', s'', ok$ and $C'', s'' \xrightarrow{k} C', s', ok$. As such, from $C'', s'' \xrightarrow{k} C', s', ok$ and the inductive hypothesis we have $C' = \text{skip}$, as required.

B.2 Soundness of BUA and FUA Rules

**Proposition 12.** For all $r, s, C, n, s', \varepsilon$, if $s \in r$, $\text{fv}(r) \cap \text{mod}(C) = \emptyset$ and $C, s \xrightarrow{n} \rightarrow, s', \varepsilon$, then $s' \in r$.

**Lemma 2.** For all $s, s', s'', C_1, C_2, C', i, j, \varepsilon$, if $C_1, s \xrightarrow{i} - , s'', ok$ and $C_2, s'' \xrightarrow{j} C', s', \varepsilon$, then there exists $n$ such that $C_1; C_2, s \xrightarrow{n} C', s', \varepsilon$.

**Proof.** Pick arbitrary $s, s', s'', C_1, C_2, C', C'', i, j, \varepsilon$, such that $C_1, s \xrightarrow{i} C'', s'', ok$ and $C_2, s'' \xrightarrow{j} C', s', \varepsilon$. We proceed by induction on $i$.

**Case $i = 0$**

From $C_1, s \xrightarrow{0} C'', s'', ok$ we know $C_1 = C'' = \text{skip}$ and $s = s''$. As such, since $C_1 = \text{skip}$ and $s = s''$, from $\text{S-SeqSkip}$ we have $C_1; C_2, s \rightarrow C_2, s'', ok$. Consequently, from $C_2, s'' \xrightarrow{j} C', s', \varepsilon$ and the definition of $\xrightarrow{j+1}$ we have $C_1; C_2, s \xrightarrow{j+1} C', s', \varepsilon$, as required.

**Case $i = k+1$**

From the definition of $C_1, s \xrightarrow{i} C'', s'', ok$ we then know there exists $C_3, s_3$ such that $C_1, s \rightarrow C_3, s_3, ok$ and $C_3, s_3 \xrightarrow{k} C'', s'', ok$. As such, from the inductive hypothesis, $C_3, s_3 \xrightarrow{k} C'', s'', ok$ and $C_2, s'' \xrightarrow{j} C', s', \varepsilon$ we know there exists $n$ such that $C_3; C_2, s_3 \xrightarrow{n} C', s', \varepsilon$. Moreover, as $C_1, s \rightarrow C_3, s_3, ok$, from $\text{S-Seq1}$ we have $C_1; C_2, s \rightarrow C_3; C_2, s_3, ok$. Consequently, as $C_1; C_2, s \rightarrow C_3; C_2, s_3, ok$ and $C_3; C_2, s_3 \xrightarrow{n} C', s', \varepsilon$, from the definition of $\xrightarrow{n+1}$ we have $C_1; C_2, s \xrightarrow{n+1} C', s', \varepsilon$, as required.

**Lemma 3.** For all $s, s', C_1, C_2, C', i, j, \varepsilon$, if $C_1, s \xrightarrow{i} C', s', er$, then $C_1; C_2, s \xrightarrow{j} C'; C_2, s', er$.

**Proof.** Pick arbitrary $s, s', C_1, C_2, C'$, such that $C_1, s \xrightarrow{i} C', s', er$. We proceed by induction on $i$.

**Case $i = 1$**

From $C_1, s \xrightarrow{1} C', s', er$ we know $C_1, s \rightarrow C', s', er$. As such, from $\text{S-Seq1}$ we have $C_1; C_2, s \rightarrow C'; C_2, s', er$. Consequently, from the definition of $\xrightarrow{i}$ we have $C_1; C_2, s \xrightarrow{1} C'; C_2, s', er$, as required.

**Case $i = k+1$**

From the definition of $C_1, s \xrightarrow{i} C', s', er$ we then know there exists $C_3, s_3$ such that $C_1, s \rightarrow C_3, s_3, ok$ and $C_3, s_3 \xrightarrow{k} C', s', er$. As such, from the inductive hypothesis and $C_3, s_3 \xrightarrow{k} C', s', er$ we know $C_3; C_2, s_3 \xrightarrow{k} C'; C_2, s', er$. Moreover, as $C_1, s \rightarrow C_3, s_3, ok$, from $\text{S-Seq1}$ we have $C_1; C_2, s \rightarrow C_3; C_2, s_3, ok$. Consequently, as $C_1; C_2, s \rightarrow C_3; C_2, s_3, ok$ and $C_3; C_2, s_3 \xrightarrow{k} C'; C_2, s', er$, from the definition of $\xrightarrow{k+1}$ we have $C_1; C_2, s \xrightarrow{k+1} C'; C_2, s', er$, as required.

**Lemma 4.** For all $n, C_1, C_2, s, s', \varepsilon$, if $C_1; C_2, s \xrightarrow{n} - , s', \varepsilon$ then either $\varepsilon = er$ and $C_1, s \xrightarrow{n} - , s', \varepsilon$, or there exists $i, j \leq n, s''$ such that $C_1, s \xrightarrow{i} - , s'', ok$ and $C_2, s'' \xrightarrow{j} - , s', \varepsilon$.

**Proof.** By induction on $n$. 
**Base case** \( n=0 \)

Pick arbitrary \( C_1, C_2, s, s', \varepsilon \) such that \( C_1; C_2, s \xrightarrow{0} -, s', \varepsilon \). This case does not arise as \( C_1; C_2, s \xrightarrow{0} -, s', \varepsilon \) would imply \( C_1; C_2 = \text{skip} \), leading to a contradiction.

**Base case** \( n=1 \) and \( \varepsilon = \text{er} \)

Pick arbitrary \( C_1, C_2, s, s', \varepsilon \) such that \( C_1; C_2, s \xrightarrow{1} -, s', \text{er} \). From the definition of \( C_1; C_2, s \xrightarrow{1} -, s', \text{er} \) we know \( C_1; C_2, s \xrightarrow{1} -, s', \text{er} \), and thus by inversion on \( C_1; C_2, s \xrightarrow{1} -, s', \text{er} \) we know \( C_1, s \xrightarrow{1} -, s', \text{er} \), as required.

**Inductive case** \( n=k+1 \)

Pick arbitrary \( C_1, C_2, s, s', \varepsilon \) such that \( C_1; C_2, s \xrightarrow{n} -, s', \varepsilon \). From the definition of \( n \to \) we then know there exist \( C', s'' \) such that \( C_1; C_2, s \to C', s'', \text{ok} \) and \( C', s'' \xrightarrow{k} -, s', \varepsilon \). From inversion on \( C_1; C_2, s \to C', s'', \text{ok} \) there are two cases to consider: 1) \( C_1 = \text{skip} \), \( C' = C_2, s'' = s \), i.e. \( C_1; C_2, s \to C_2, s, \text{ok} \); or 2) there exists \( C_1' \) such that \( C_1, s \to C_1', s'', \text{ok} \) and \( C' = C_1'; C_2 \).

In case (1), by definition we have \( C_1, s \xrightarrow{0} \text{skip}, s'', \text{ok} \). Moreover, as \( C' = C_2 \), from \( C', s'' \xrightarrow{k} -, s', \varepsilon \) we have \( C_1, s'' \xrightarrow{k} -, s', \varepsilon \). That is, as \( 0 \leq n \) and \( k \leq n \), we have \( C_1, s \xrightarrow{0} \text{skip}, s'', \text{ok} \) and \( C_2, s'' \xrightarrow{k} -, s', \varepsilon \), as required.

In case (2), as \( C' = C_1'; C_2 \) and \( C', s'' \xrightarrow{k} -, s', \varepsilon \), from the inductive hypothesis we know either a) \( \varepsilon = \text{er} \) and \( C_1', s'' \xrightarrow{k} -, s', \varepsilon \); or b) there exist \( i, j \leq k, s_2 \) such that \( C_1', s'' \xrightarrow{i} -, s_2, \text{ok} \) and \( C_2, s_2 \xrightarrow{j} -, s', \varepsilon \).

In case (2.a), as \( \varepsilon = \text{er} \), \( C_1, s \to C_1', s'', \text{ok} \) and \( C_1', s'' \xrightarrow{k} -, s', \varepsilon \), from the definition of \( n \) we have \( C_1, s \xrightarrow{n} -, s', \text{er} \), as required.

In case (2.b), as \( i \leq k \), \( C_1, s \to C_1', s'', \text{ok} \) and \( C_1', s'' \xrightarrow{i} -, s_2, \text{ok} \), from the definition of \( i+1 \) we know there exists \( m=i+1 \leq k+1 = n \) such that \( C_1, s \xrightarrow{m} -, s_2, \text{ok} \). Moreover, we also know there exists \( j \leq k < n \) such that \( C_2, s_2 \xrightarrow{j} -, s', \varepsilon \). That is, we know there exist \( m, j \leq n \) such that \( C_1, s \xrightarrow{m} -, s_2, \text{ok} \) and \( C_2, s_2 \xrightarrow{j} -, s', \varepsilon \), as required.

**Lemma 5.** For all \( n, C, s, s', \varepsilon \), if \( C^*; C, s \xrightarrow{n} -, s', \varepsilon \) then there exists \( m \) such that \( C; C^*, s \xrightarrow{m} -, s', \varepsilon \).

**Proof.** By strong induction on \( n \).

**Base case** \( n=0 \)

Pick arbitrary \( C, s, s', \varepsilon \) such that \( C^*; C, s \xrightarrow{0} -, s', \varepsilon \). This case does not arise as \( C^*; C, s \xrightarrow{0} -, s', \varepsilon \) would imply \( C^*; C = \text{skip} \), leading to a contradiction.

**Base case** \( n=1 \)

Pick arbitrary \( C, s, s', \varepsilon \) such that \( C^*; C, s \xrightarrow{1} -, s', \varepsilon \). This case also does not arise. Specifically, from \( C^*; C, s \xrightarrow{1} -, s', \varepsilon \) we know that \( \varepsilon = \text{er} \) and \( C^*; C, s \to -, s', \varepsilon \), i.e. \( C^*; C, s \to -, s', \text{er} \). By inversion, the only transition that could apply is that of \text{S-Seq1}, meaning that there exists \( C' \) such that \( C^*, s \to -, s', \text{er} \). However, by inversion, no transition in Fig. 8 allows us to take an erroneous transition of the form \( C^*, s \to -, s', \text{er} \).

**Inductive case** \( n=k+1 \)

Pick arbitrary \( C, s, s', \varepsilon \) such that \( C^*; C, s \xrightarrow{n} C', s', \varepsilon \). From \( C^*; C, s \xrightarrow{n} -, s', \varepsilon \) we know there exists \( s'', C' \) such that \( C^*; C, s \to C', s'', \text{ok} \) and \( C', s'' \xrightarrow{k} -, s', \varepsilon \). From \( C^*; C, s \to C', s'', \text{ok} \) and by
inversion through $S_{Seq1}$ we know there exists $C'$ such that $C^*, s \rightarrow C', s'', ok$ and $C' = C'_1; C$.

By inversion on $C^*, s \rightarrow C'_1; s''$, ok there are two cases to consider: 1) Through $S_{Loop0}$ we have $C'_1 = \text{skip}$ and $s'' = s$, i.e. $C^*, s \rightarrow \text{skip}, s, ok$; or 2) Through $S_{Loop}$ we have $C'_1 = C; C^*$ and $s'' = s$, i.e. $C^*, s \rightarrow C; C^*, s, ok$.

In case (1), from $C', s'' \xrightarrow{k} -, s', \epsilon, C' = C'_1; C$ and the assumption of case we have skip; $C$, $s \xrightarrow{k} -, s', \epsilon$. As such, from the definition of $\xrightarrow{k}$ and inversion we know the cases where $k=0$ or $k=1 \land \epsilon = er$
do not arise, and that skip; $C, s \rightarrow C, s, ok$ and $C, s \xrightarrow{k-1} -, s', \epsilon$. There are now to subcases to consider:
a) $\epsilon = ok$; or b) $\epsilon = er$.

In case (1.a), we have $C, s \xrightarrow{k-1} -, s', ok$. Moreover, from $S_{Loop0}$ we have $C^*, s' \rightarrow \text{skip}, s', ok$, and thus since by definition we also have skip, $s' \xrightarrow{0} \text{skip}, s', ok$, by definition we have $C^*, s' \xrightarrow{1} \text{skip}, s', ok$.

As such, from $C, s \xrightarrow{k-1} -, s', ok; C^*, s' \xrightarrow{1} \text{skip}, s', ok$ and Lemma 2 we know there exists $m$ such that $C; C^*, s \xrightarrow{m} -, s', ok$; i.e. $C; C^*, s \xrightarrow{m} -, s', \epsilon$, as required.

In case (1.b), we have $C, s \xrightarrow{k-1} -, s', er$. As such, from Lemma 3 we have $C; C^*, s \xrightarrow{k-1} -, s', er$, i.e. there exists $m$ such that $C; C^*, s \xrightarrow{m} -, s', \epsilon$, as required.

In case (2), as $C'_1 = C; C^*, s'' = s, C' = C'_1; C$ and $C$, $s' \xrightarrow{k} -, s', \epsilon$, we know $C; C^*; C, s \xrightarrow{k} -, s', \epsilon$.

From Lemma 4 we then know there are two cases to consider: a) $\epsilon = er$ and $C, s \xrightarrow{k} -, s', \epsilon$; or b) there exists $i, j \leq n$, $s_1$ such that $C, s \xrightarrow{k} -, s_1, ok$ and $C^*; C, s_1 \xrightarrow{j} -, s', \epsilon$.

In case (2.a), as $\epsilon = er$ and $C, s \xrightarrow{k} -, s', \epsilon$, from Lemma 3 we have $C; C^*, s \xrightarrow{k} -, s', \epsilon$, as required.

In case (2.b), as $j \leq k$ and $C^*; C, s_1 \xrightarrow{j} -, s', \epsilon$, from the inductive hypothesis we know there exists a such that $C; C^*, s_1 \xrightarrow{j} -, s', \epsilon$. Moreover, from $S_{Loop}$ we have $C^*, s_1 \rightarrow C; C^*, s_1, ok$.

As such, from $C; C^*, s_1 \xrightarrow{j} -, s', \epsilon$ and the definition of $\xrightarrow{a+1}$ we have $C^*, s_1 \xrightarrow{j} -, s', \epsilon$. Consequently, since from the assumption of case (2.b) we also have $C, s \xrightarrow{i} -, s_1, ok$, from Lemma 2 we know there exists $m$ such that $C; C^*, s \xrightarrow{m} -, s', \epsilon$, as required.

\[\square\]

Lemma 6. For all $p, C$, if $\forall n \in \mathbb{N}$. $\models_B [p(n)] C \left[ok: p(n+1)\right]$, then $\forall k, i \in \mathbb{N}$. $\models_B [p(i)] C^* \left[ok: p(i+k)\right]$.

Proof. Pick arbitrary $p, C$ such that $\forall n \in \mathbb{N}$. $\models_B [p(n)] C \left[ok: p(n+1)\right]$. We proceed by induction on $k$.

Base case $k=0$

Pick an arbitrary $i \in \mathbb{N}$. We are then required to show $\models_B [p(i)] C^* \left[ok: p(i)\right]$. Pick an arbitrary $s \in p(i)$. From $S_{Loop0}$ we have $C^*, s \rightarrow \text{skip}, s, ok$.

As such, as we have skip, $s \xrightarrow{0} \text{skip}, s, ok$ (from the definition of $\xrightarrow{0}$), by definition we have $C^*, s \xrightarrow{1} \text{skip}, s, ok$. Consequently, we have $s \in p(i)$ and $C^*, s \xrightarrow{1} \text{skip}, s, ok$, as required.

Inductive case $k=j+1$

Pick an arbitrary $i \in \mathbb{N}$ and $s \in p(i)$. From $\forall n \in \mathbb{N}$. $\models_B [p(n)] C \left[ok: p(n+1)\right]$ we know $\models_B [p(i)] C \left[ok: p(i+1)\right]$ holds, and thus since $s \in p(i)$, from the definition of $\models_B$ we then know there exists $s'' \in p(i+1), m$ such that $C, s \xrightarrow{m} -, s'', ok$. 
On the other hand, from the inductive hypothesis we know \( \forall a \in \mathbb{N}. \ \vdash_B \ [p(a)] \) \( \mathbf{C}^* \ [ok \ : p(a+j)] \).

As such, from the inductive hypothesis we have \( \vdash_B \ [p(i+1)] \) \( \mathbf{C}^* \ [ok \ : p(i+1+j)] \), i.e. \( \vdash_B \ [p(i+1)] \) \( \mathbf{C}^* \ [ok \ : p(i+k)] \). Consequently, since \( s'' \in p(i+1) \), from the definition of \( \vdash_B \) we know there exists \( s' \in p(i+k), b \) such that \( \mathbf{C}^*, s', \stackrel{b}{\rightarrow}, s', ok \). Therefore, from Lemma 2, \( C, s \ x \rightarrow, s', ok \) and \( \mathbf{C}^*, s', \stackrel{b}{\rightarrow}, s', ok \) we know there exists \( c \) such that \( \mathbf{C} \mathbf{C}^*, s', \stackrel{c}{\rightarrow}, s', ok \).

Furthermore, from S-LOOP we simply have \( \mathbf{C}^*, s, \rightarrow \rightarrow \mathbf{C}^*, s, ok \). As such, since we also have \( \mathbf{C} \mathbf{C}^*, s, \stackrel{\epsilon}{\rightarrow}, s', ok \), from the definition of \( \stackrel{\epsilon+1}{\rightarrow} \) we have \( \mathbf{C}^*, s, \stackrel{\epsilon+1}{\rightarrow}, s', ok \). That is, we have \( s' \in p(i+k) \) such that \( \mathbf{C}^*, s, \stackrel{\epsilon+1}{\rightarrow}, s', ok \), as required.

\[ \square \]

**Lemma 7 (BUA soundness).** For all \( p, C, q, \epsilon \), if \( \vdash_B \ [p] \) \( C \ [\epsilon : q] \) can be proven using the proof rules in Fig. 2, then \( \vdash_B \ [p] \) \( C \ [\epsilon : q] \) holds.

**Proof.** By induction on the structure of rules in Fig. 2.

**Case Skip**

Pick arbitrary \( p \) such that \( \vdash_B \ [p] \) skip \( [ok : p] \). Pick an arbitrary \( s \in p \). From the semantics of skip we then have skip, \( s \stackrel{0}{\rightarrow} \) skip, \( s \) and \( s \) ok, as required.

**Case Assign**

Pick arbitrary \( p \) such that \( \vdash_B \ [p] \) \( x := e \ [ok : \exists y. p[y/x] \land x = e[y/x]] \). Pick an arbitrary \( s \in p \). Let \( s(x) = v_x, s(e) = v_e \) and \( s' = s[x \mapsto v_e] \). From S-Assign we then have \( x := e, s \rightarrow \) skip, \( s', ok \). As such, since we also have skip, \( s \stackrel{0}{\rightarrow} \) skip, \( s', ok \), by definition we have \( x := e, s \rightarrow \) skip, \( s', ok \).

As \( s(x) = v_x \) and \( s(e) = v_e \), by definition we have \( s(e[v_x/x]) = v_e \) and \( s'(e[v_x/x]) = v_e \). As \( s \in p \) and \( s(x) = v_x \), we also have \( s \in p[v_x/x] \). Thus, as \( s' = s[x \mapsto v_e] \) and \( s \in p[v_x/x] \), we also have \( s' \in p[v_x/x] \). Similarly, as \( s'(e[v_x/x]) = v_e \) and \( s' = s[x \mapsto v_e] \) (i.e. \( s'(x) = v_e \)), we have \( s' \in x = e[v_x/x] \). That is, we have \( s' \in p[v_x/x] \land x = e[v_x/x] \). Let \( s'' = s'[y \mapsto v_x] \). Consequently, as \( s' \in p[v_x/x] \land x = e[v_x/x] \), we also have \( s'' \in p[y/x] \land x = e[y/x] \). As such, since \( s'' \in p[y/x] \land x = e[y/x] \) and \( s'' \in s'[y \mapsto v_x] \), by definition we have \( s' \in \exists y. p[y/x] \land x = e[y/x] \).

Therefore, we have \( x := e, s \stackrel{1}{\rightarrow} \) skip, \( s', ok \) and \( s' \in \exists y. p[y/x] \land x = e[y/x] \), as required.

**Case Assume**

Pick arbitrary \( p, B \) such that \( \vdash_B \ [p \land B] \) assume(B) \( [ok : p \land B] \). Pick an arbitrary \( s \in p \land B \). By definition we then know \( s(B) = true \). From S-Assume we then have assume(B), \( s \rightarrow \) skip, \( s \) ok.

As such, since we also have skip, \( s \stackrel{0}{\rightarrow} \) skip, \( s \) ok, by definition we have assume(B), \( s \stackrel{1}{\rightarrow} \) skip, \( s \) ok.

Consequently, we have \( s \in p \land B \) and assume(B), \( s \rightarrow \) skip, \( s \) ok, as required.

**Case Error**

Pick arbitrary \( p \) such that \( \vdash_B \ [p] \) error \( [er : p] \). Pick an arbitrary \( s \in p \). From S-Error we then have error, \( s \rightarrow \) skip, \( s \) er. As such, by definition we have error, \( s \stackrel{1}{\rightarrow} \) skip, \( s \) er. Consequently, we have \( s \in p \) and error, \( s \stackrel{1}{\rightarrow} \) skip, \( s \) er, as required.

**Case SEQ**

Pick arbitrary \( p, q, r, C_1, C_2, \epsilon \) such that \( \vdash_B \ [p] \) \( C_1 \ [ok : r] \) and \( \vdash_B \ [r] \) \( C_2 \ [\epsilon : q] \). Pick an arbitrary \( s \in p \). From \( \vdash_B \ [p] \) \( C_1 \ [ok : r] \) and the inductive hypothesis we then know there exists \( s'' \in r, i \) such that \( C_1, s \stackrel{i}{\rightarrow}, s'', ok \). Moreover, as \( s'' \in r, i \), from \( \vdash_B \ [r] \) \( C_2 \ [\epsilon : q] \) and the inductive hypothesis
we have there exists \( s' \in q, j \) such that \( C_2, s'' \overset{j}{\rightarrow} s', \epsilon \). As \( C_1, s \overset{i}{\rightarrow} s'', \text{ok} \) and \( C_2, s'' \overset{j}{\rightarrow} s', \epsilon \), from Lemma 2 we know there exists \( n \) such that \( C_1; C_2, s^n \overset{n}{\rightarrow} s', \epsilon \). That is, there exists \( s' \in q, n \) such that \( C_1; C_2, s^n \overset{n}{\rightarrow} s', \epsilon \), as required.

Case SeqEr

Pick arbitrary \( p, q, C_1, C_2 \) such that \( \vdash_B [p] C_1; C_2 \ [er: q] \). Pick an arbitrary \( s \in p \). From the \( \vdash_B [p] C_1 \ [er: q] \) premise and the inductive hypothesis we then know there exists \( s' \in q, i \) such that \( C_1, s \overset{i}{\rightarrow} s', \text{er} \). As such, from Lemma 3 we know \( C_1; C_2, s \overset{i}{\rightarrow} s', \text{er} \). That is, there exists \( s' \in q \) such that \( C_1; C_2, s \overset{i}{\rightarrow} s', \text{er} \), as required.

Case Choice

Pick arbitrary \( p, q, C_1, C_2, \epsilon \) and \( i \in \{1, 2\} \) such that \( \vdash_B [p] C_1 + C_2 \ [\epsilon : q] \). Pick an arbitrary \( s \in p \). From the \( \vdash_B [p] C_1 \ [\epsilon : q] \) premise and the inductive hypothesis we then know there exists \( s' \in q, j \) such that \( C_1, s \overset{j}{\rightarrow} s', \epsilon \). Moreover, from S-Choice we have \( C_1 + C_2, s \overset{i}{\rightarrow} C_i, s, \text{ok} \). As such, from the definition of \( \overset{j+1}{\rightarrow} \) we have \( C_1 + C_2, s \overset{j+1}{\rightarrow} s', \epsilon \). That is, there exists \( s' \in q \) such that \( C_1 + C_2, s \overset{j+1}{\rightarrow} s', \epsilon \), as required.

Case Loop

Pick arbitrary \( p, C \) such that \( \vdash_B [p] C^* \ [\text{ok}: p] \). Pick an arbitrary \( s \in p \). From S-Loop0 we have \( C^*, s \overset{j}{\rightarrow} \text{skip}, s, \text{ok} \). As such, as we have skip, \( s \overset{0}{\rightarrow} \text{skip}, s, \text{ok} \) (from the definition of \( 0 \)), by definition we have \( C^*, s \overset{1}{\rightarrow} \text{skip}, s, \text{ok} \). Consequently, we have \( s \in p \) and \( C^*, s \overset{1}{\rightarrow} \text{skip}, s, \text{ok} \), as required.

Case Loop-Subvariant

Pick arbitrary \( p, C, k \) such that \( \vdash_B \begin{array}{c} p(0) \end{array} C^* \ [\text{ok}: p(k)] \). From the \( \forall n \in \mathbb{N} \). \( \vdash_B \begin{array}{c} p(n) \end{array} C \ [\text{ok}: p(n+1)] \) premise and the inductive hypothesis we have \( \forall n \in \mathbb{N} \). \( \vdash_B \begin{array}{c} p(n) \end{array} C \ [\text{ok}: p(n+1)] \). Consequently, from Lemma 6 we have \( \vdash_B \begin{array}{c} p(0) \end{array} C^* \ [\text{ok}: p(k)] \), as required.

Case Local

Pick arbitrary \( p, C, q, \epsilon \) such that \( \vdash_B \exists x. \ p \) local \( x \) in \( C \ [\epsilon : \exists x. \ q] \). Pick an arbitrary \( s \in \exists x. \ p \); i.e. there exists \( v, s_p \) such that \( s_p = s[x \mapsto v] \) and \( s_p \in p \). From the \( \vdash_B [p] C \ [\epsilon : q] \) premise and the inductive hypothesis we know there exists \( s_q \in q \) such that \( C, s_p \overset{n}{\rightarrow} s_q, \epsilon \). From S-Local we have local \( x \) in \( C, s \rightarrow C; \text{end}(x, s(x)), s_p \). There are now two cases to consider: 1) \( \epsilon = \text{ok} \); or 2) \( \epsilon = \text{er} \).

In case (1), let \( s'' = s_q[x \mapsto s(x)] \). From S-LocalEnd we then have \( \text{end}(x, s(x)), s_q \rightarrow \text{skip}, s'' \).

From the definition of \( 0 \) we have \( \text{skip}, s'' \overset{0}{\rightarrow} \text{skip}, s'', \text{ok} \), and thus since we have \( \text{end}(x, s(x)), s_q \rightarrow \text{skip}, s'' \), from the definition of \( 1 \) we have \( \text{end}(x, s(x)), s_q \overset{1}{\rightarrow} \text{skip}, s'' \). Consequently, since we
also have $C$, $s_p \xrightarrow{n}, s_q, \epsilon$, from Lemma 2 we know there exists $m$ such that $C; \text{end}(x, s(x)), s_p \xrightarrow{m}$ skip, $s''$, ok. On the other hand, since we have local $x$ in $C$, $s \xrightarrow{m+1}$, by definition of $\xrightarrow{n+1}$ we also have local $x$ in $C$, $s \xrightarrow{m+1}$ skip, $s''$, ok. Finally, as $s_q \in q$ and $s'' = s_q[x \mapsto s(x)]$, by definition we also have $s'' \in \exists x. q$, as required.

In case (2), from $C$, $s_p \xrightarrow{n}, s_q, \epsilon$ and Lemma 3 we have $C; \text{end}(x, s(x)), s_p \xrightarrow{n}, s_q, \epsilon$. On the other hand, since we have local $x$ in $C$, $s \xrightarrow{n+1}$, $s_q, \epsilon$. Finally, as $s_q \in q$, by definition we also have $s_q \in \exists x. q$, as required.

**Case SUBST**

Pick arbitrary $p, C, q, y$ such that $y \notin \text{fv}(p, C, q)$ and $(\vdash_B [p] C [e : q])[(y/x)], \text{i.e. } (\vdash_B [p[y/x]] C[y/x] [e : q[y/x]])$. Pick an arbitrary $s \in p[y/x]$ and let $s_p = s[x \mapsto s(y)]$. We then have $s_p \in p$ and thus from the $\vdash_B [p] C [e : q]$ premise and the inductive hypothesis we know there exists $s_q \in q, n$ such that $C, s_p \xrightarrow{n}, s_q, \epsilon$. Let $s' = s_q[y \mapsto x]$; as $s_q \in q$, we then have $s' \in q[y/x]$. As such, from the semantics we also have $C[y/x], s \xrightarrow{n}, s', \epsilon$, as required.

**Case DISJ**

Pick arbitrary $p_1, p_2, q_1, q_2, C$ such that $\vdash_B [p_1 \lor p_2] C [e : q_1 \lor q_2]$. Pick an arbitrary $s \in p_1 \lor p_2$. There are then two cases to consider: 1) $s \in p_1$; or 2) $s \in p_2$.

In case (1), from the $\vdash_B [p_1] C [e : q_1]$ premise and the inductive hypothesis we know there exists $s' \in q_1, n$ such that $C, s \xrightarrow{n}, s', \epsilon$. That is, there exists $s' \in q_1 \lor q_2$ and $n$ such that $C, s \xrightarrow{n}, s', \epsilon$, as required. The proof of case (2) is analogous and omitted.

**Case Constancy**

Pick arbitrary $p, q, r, C$ such that $\vdash_B [p \land r] C [e : q \land r]$. Pick an arbitrary $s \in p \land r$. That is, $s \in p$ and $s \in r$. From the $\vdash_B [p] C [e : q]$ premise and the inductive hypothesis we know there exists $s' \in q, n$ such that $C, s \xrightarrow{n}, s', \epsilon$. As such, from the $\text{fv}(r) \cap \text{mod}(C) = \emptyset$ premise, Prop. 12 and since $s \in r$, we know $s' \in r$. Therefore, we have $s' \in q$ and $s' \in r$ and thus $s' \in q \land r$. That is, there exists $s' \in q \land r$ and $n$ such that $C, s \xrightarrow{n}, s', \epsilon$, as required.

**Case ConsF**

Pick arbitrary $p, q, C$ such that $\vdash_B [p] C [e : q]$. Pick an arbitrary $s \in p$. From the $p \subseteq p'$ premise we then have $s \in p'$. Moreover, from the $\vdash_B [p'] C [e : q']$ and the inductive hypothesis we know there exists $s' \in q'$ and $n$ such that $C, s \xrightarrow{n}, s', \epsilon$. As $q' \subseteq q$ and $s' \in q'$, we also have $s' \in q$. That is, there exists $s' \in q$ and $n$ such that $C, s \xrightarrow{n}, s', \epsilon$, as required.

**Case DISJTrack**

Pick arbitrary $P_1, P_2, Q_1, Q_2, C$ such that $\vdash_B [P_1 \uplus P_2] C [e : Q_1 \uplus Q_2]$. Pick an arbitrary $i \in \text{dom}(P_1 \uplus P_2)$ and $s \in (P_i \uplus P_2)(i)$. We then know that either $i \in \text{dom}(P_1)$ or $i \in \text{dom}(P_2)$. Without loss of generality, let us assume $i \in \text{dom}(P_1)$.

As $s \in (P_i \uplus P_2)(i)$ and $i \in \text{dom}(P_1)$, we then have $s \in P_1(i)$. From the $\vdash_B [P_1] C [e : Q_1]$, the definition of merged triples premise and the inductive hypothesis we know there exists $s' \in Q_1(i), n$ such that $C, s \xrightarrow{n}, s', \epsilon$. That is, there exists $s' \in (Q_1 \uplus Q_2)(i)$ and $n$ such that $C, s \xrightarrow{n}, s', \epsilon$, as required.
Case **Cons**

Pick arbitrary $P, Q, C, I$ such that $\vdash_B [P \downarrow I] C [\epsilon : Q \downarrow I]$. Pick an arbitrary $i \in \text{dom}(P \downarrow I)$; that is, from the $I \subseteq \text{dom}(P)$ we know $i \in \text{dom}(P) \cap I$, i.e. $i \in \text{dom}(P)$ and $i \in I$. Pick an arbitrary $s \in P(i)$. From the $\vdash_B [P] C [\epsilon : Q]$ premise the definition of merged triples and the inductive hypothesis we know there exists $s' \in Q(i)$ and $n$ such that $C, s \xrightarrow{n} - , s', \epsilon$. As $i \in I$ and $i \in \text{dom}(Q)$, we know $i \in \text{dom}(Q \downarrow I)$. That is, there exists $i \in \text{dom}(Q \downarrow I), s' \in (Q \downarrow I)(i)$ and $n$ such that $C, s \xrightarrow{n} - , s', \epsilon$, as required.

**Lemma 8** (FUA soundness). For all $p, C, q, \epsilon$, if $\vdash_F [p] C [\epsilon : q]$ can be proven using the proof rules in Fig. 2, then $\models_F [p] C [\epsilon : q]$ holds.

**Proof.** By induction on the structure of rules in Fig. 2.

Cases **Skip, Assign, Error, Seq, SeqEr, Choice, Loop0, Loop, Loop-Subvariant, Disj, Constancy, ConsB, Subst**

The proof of these cases is as given by O’Hearn [23].

Case **Local**

Pick arbitrary $p, C, q, \epsilon$ such that $\vdash_F [\exists x. p]$ local $x$ in $C [\epsilon : \exists x. q]$. Pick an arbitrary $s' \in \exists x. q$; i.e. there exists $v, s_q$ such that $s_q = s'[x \mapsto v]$ and $s_q \in q$. From the $\vdash_F [p] C [\epsilon : q]$ premise and the inductive hypothesis we know there exists $s_p \in p$ and $n$ such that $C, s_p \xrightarrow{n} - , s_q, \epsilon$. From S-Local we have local $x$ in $C$, $s_p \rightarrow C; \text{end}(x, s_p(x))$, $s_p$. There are two cases to consider: 1) $\epsilon = \text{ok}$; or 2) $\epsilon = \text{er}$.

In case (1), let $s'' = s_q[x \mapsto s_p(x)]$. From S-LocalEnd we then have $\text{end}(x, s_p(x))$, $s_q \rightarrow \text{skip}, s''$.

From the definition of $\rightarrow 0$ we have skip, $s'' \rightarrow 0$ skip, $s''$, ok, and thus since we have $\text{end}(x, s_p(x))$, $s_q \rightarrow$ skip, $s''$, from the definition of $\rightarrow 1$ we have $\text{end}(x, s_p(x))$, $s_q \rightarrow \text{skip}, s''$. Consequently, since we also have $C, s_p \xrightarrow{n} - , s_q, \epsilon$, from Lemma 2 we know there exists $m$ such that $C; \text{end}(x, s_p(x)), s_p \xrightarrow{m} \text{skip}, s'', \text{ok}$. On the other hand, since we have local $x$ in $C$, $s_p \rightarrow C; \text{end}(x, s_p(x)), s_p$, by definition of $\xrightarrow{n+1}$ we also have local $x$ in $C$, $s_p \xrightarrow{m+1} \text{skip}, s'', \text{ok}$. Finally, as $s_p \in p$, by definition we also have $s_p \in \exists x. p$, as required.

In case (2), from $C$, $s_p \xrightarrow{n} - , s_q, \epsilon$ and Lemma 3 we have $C; \text{end}(x, s_p(x)), s_p \xrightarrow{n} - , s_q, \epsilon$. On the other hand, since we have local $x$ in $C$, $s_p \rightarrow C; \text{end}(x, s_p(x)), s_p$, by definition of $\xrightarrow{n+1}$ we also have local $x$ in $C$, $s_p \xrightarrow{n+1} - , s_q, \epsilon$. Finally, as $s_p \in p$, by definition we also have $s_p \in \exists x. p$, as required.

Case **Assume**

Pick arbitrary $p, B$ such that $\vdash_F [p \land B]$ assume$(B) [\text{ok} : p \land B]$. Pick an arbitrary $s \in p \land B$. By definition we then know $s(B) = \text{true}$. From S-Assume we then have assume$(B), s \rightarrow \text{skip}, s, \text{ok}$.

As such, since we also have skip, $s \xrightarrow{0} \text{skip}, s, \text{ok}$, by definition we have assume$(B), s \xrightarrow{1} \text{skip}, s, \text{ok}$.

Consequently, we have $s \in p \land B$ and assume$(B), s \xrightarrow{1} \text{skip}, s, \text{ok}$, as required.

Case **Disj**

Pick arbitrary $P_1, P_2, Q_1, Q_2, C$ such that $\vdash_F [P_1 \lor P_2] C [\epsilon : Q_1 \lor Q_2]$. Pick an arbitrary $i \in \text{dom}(Q_1 \lor Q_2)$ and $s' \in (Q_1 \lor Q_2)(i)$. We then know that either $i \in \text{dom}(Q_1)$ or $i \in \text{dom}(Q_2)$. Without loss of generality, let us assume $i \in \text{dom}(Q_1)$.

As $s' \in (Q_1 \lor Q_2)(i)$ and $i \in \text{dom}(Q_1)$, we then have $s' \in Q_1(i)$. From the $\vdash_F [P_1] C [\epsilon : Q_1]$ premise, the definition of merged triples and the inductive hypothesis we know there exists
\[ s \in P_i(i), n \text{ such that } C, s \to^n n, s', \epsilon. \text{ That is, there exists } s \in (P_1 \cup P_2(i)) \text{ and } n \text{ such that } C, s \to^n n, s', \epsilon, \text{ as required.} \]

**Case Cons**

Pick arbitrary \( P, Q, C, I \) such that \( \tau_B [P \downarrow I] C [\epsilon : Q \downarrow I] \). Pick an arbitrary \( i \in \text{dom}(Q \downarrow I) \); that is, from the \( I \subseteq \text{dom}(P) \) we know \( i \in \text{dom}(Q) \cap I \), i.e. \( i \in \text{dom}(Q) \) and \( i \in I \). Pick an arbitrary \( s' \in Q(i) \).

From the \( \tau_F [P] C [\epsilon : Q] \) premise of the definition of merged triples and the inductive hypothesis we know there exists \( s \in P(i) \) and \( n \) such that \( C, s \to^n n, s', \epsilon \). As \( i \in I \) and \( i \in \text{dom}(P) \), we know \( i \in \text{dom}(P \downarrow I) \). That is, there exists \( i \in \text{dom}(P \downarrow I) \), \( s \in (P \downarrow I)(i) \) and \( n \) such that \( C, s \to^n n, s', \epsilon \), as required. \( \square \)

**Theorem 13** (Soundness). For all \( P, C, q, \epsilon \), if \( \tau_+ [p] C [\epsilon : q] \) can be proven using the proof rules in Fig. 2, then \( \models_+ [p] C [\epsilon : q] \) holds.

**Proof.** Follows immediately from Lemma 7 and Lemma 8. \( \square \)

**B.3 Soundness of Divergence Rules**

In what follows, we write \( C, s \leadsto^+ C', s', \epsilon \) for \( \exists n. C, s \leadsto^n C', s', \epsilon \).

**Lemma 9.** For all \( C, s, C', s', \epsilon, n \), if \( n > 0 \) and \( C, s \to^n C', s', \epsilon \), then \( C, s \leadsto^n C', s', \epsilon \).

**Proof.** By induction on \( n \).

**Base case** \( n = 1 \)

Pick arbitrary \( C, C', s, C', s', \epsilon \) such that \( C, s \to C', s', \epsilon \). From the definition of \( \to \) there are then two cases to consider: 1) \( \epsilon = \text{er} \) and \( C, s \to C', s', \text{er}; \) or 2) \( \epsilon = \text{ok} \), \( C' = \text{skip and } C, s \to C', s', \text{ok} \).

In case (1), from the definition of \( \to^1 \) we also have \( C, s \to^1 C', s', \text{er} \), as required. In case (2), from the definition of \( \to^1 \) we also have \( C, s \to^1 C', s', \text{ok} \), as required.

**Inductive case** \( n = k+1 \) with \( k > 0 \)

Pick arbitrary \( C, C', s, C', s', \epsilon \) such that \( C, s \to^n C', s', \epsilon \). From the definition of \( \to^n \), we know there exists \( C'', s'' \) such that \( C, s \to C'', s'', \text{ok} \) and \( C'', s'' \to^n C', s', \epsilon \). From \( C'', s'' \to^n C', s', \epsilon \) and the inductive hypothesis we have \( C'', s'' \to^k C', s', \epsilon \). As such, from \( C, s \to C'', s'', \text{ok} \) and the definition of \( \to^n \) we have \( C, s \to^n C', s', \epsilon \), as required. \( \square \)

**Lemma 10.** For all \( n, C_1, C_2, C'_1, s, C', s', \epsilon \), if \( C_1, s \to^n C_1', s', \epsilon \), then \( C_1; C_2, s \to^n C_1'_1; C_2, s', \epsilon \).

**Proof.** By induction on \( n \).

**Base case** \( n = 1 \)

Pick arbitrary \( C_1, C_2, C'_1, s, C', s', \epsilon \) such that \( C_1, s \to^1 C'_1, s', \epsilon \). From the definition of \( \to^1 \) we then know \( C_1, s \to C'_1, s', \epsilon \). As such, from S-Seq1 we have \( C_1; C_2, s \to C'_1; C_2, s', \epsilon \), and thus by definition of \( \to^1 \) we have \( C_1; C_2, s \to^1 C'_1; C_2, s', \epsilon \), as required.

**Inductive case** \( n = k+1 \)

Pick arbitrary \( C_1, C_2, C'_1, s, C', s', \epsilon \) such that \( C_1, s \to^n C_1'_1, s', \epsilon \). From the definition of \( \to^n \) we then know there exists \( C'', s'' \) such that \( C_1, s \to C'', s'', \text{ok} \) and \( C'', s'' \to^k C'_1, s', \epsilon \). From \( C_1, s \to C'', s'', \text{ok} \) and S-Seq1 we have \( C_1; C_2, s \to C''; C_2, s'', \text{ok} \). From \( C'', s'' \to^k C'_1, s', \epsilon \) and the inductive hypothesis we have \( C''; C_2, s'' \to^k C'_1; C_2, s', \epsilon \). As such, since we have \( C_1; C_2, s \to C''; C_2, s'', \text{ok} \)
and \( C''; C_2, s'' \leadsto^k C'_1; C_2, s', \epsilon \), from the definition of \( \leadsto^n \) we have \( C_1; C_2, s \leadsto^n C'_1; C_2, s', \epsilon \), as required.

**Lemma 11.** For all \( s, s', s'', C_1, C_2, C', i, j, \epsilon \), if \( C_1, s \xrightarrow{i} s'' \) and \( C_2, s'' \leadsto^j C', s', \epsilon \), then there exists \( n \) such that \( C_1; C_2, s \leadsto^n C', s', \epsilon \).

**Proof.** Pick arbitrary \( s, s', s'', C_1, C_2, C', C'' \), \( i, j, \epsilon \), such that \( C_1, s \xrightarrow{i} C'', s'', \text{ok} \) and \( C_2, s'' \leadsto^j C', s', \epsilon \). We proceed by induction on \( i \).

**Case** \( i = 0 \)

From \( C_1, s \rightarrow C'', s'', \text{ok} \) we know \( C_1 = C'' = \text{skip} \) and \( s = s'' \). As such, since \( C_1 = \text{skip} \) and \( s = s'' \), from S-SEQSKIP we have \( C_1; C_2, s \rightarrow C_2, s'', \text{ok} \). Consequently, from \( C_2, s'' \leadsto^j C', s', \epsilon \) and the definition of \( \leadsto^{n+1} \) we have \( C_1; C_2, s \leadsto^{n+1} C', s', \epsilon \), as required.

**Case** \( i = k+1 \)

From the definition of \( C_1, s \rightarrow C'', s'', \text{ok} \) we then know there exists \( C_3, s_3 \) such that \( C_1, s \rightarrow C_3, s_3, \text{ok} \) and \( C_3, s_3 \rightarrow C'', s'', \text{ok} \). As such, from the inductive hypothesis, \( C_3, s_3 \rightarrow C'', s'', \text{ok} \) and \( C_2, s'' \leadsto^j C', s', \epsilon \). Moreover, as \( C_1, s \rightarrow C_3, s_3, \text{ok} \), from S-SEQ1 we have \( C_1; C_2, s \rightarrow C_3; C_2, s_3, \text{ok} \). Consequently, as \( C_1; C_2, s \rightarrow C_3; C_2, s_3, \text{ok} \) and \( C_3; C_2, s_3 \rightarrow C', s', \epsilon \), from the definition of \( \leadsto^{n+1} \) we have \( C_1; C_2, s \leadsto^{n+1} C', s', \epsilon \), as required. □

**Lemma 12.** For all \( i, j, C, C', C'', s, s', s'', \epsilon \), if \( C, s \leadsto^1 C'', s'', \text{ok} \) and \( C', s', \text{ok} \), then \( C, s \leadsto^{i+j} C', s', \epsilon \).

**Proof.** By induction on \( i \).

**Base case** \( i = 1 \)

Pick arbitrary \( j, C, C', C'', s, s', s'', \epsilon \) such that \( C, s \leadsto^1 C'', s'', \text{ok} \) and \( C', s', \text{ok} \), and thus from \( C'', s'' \leadsto^j C', s', \epsilon \) and the definition of \( \leadsto^{i+j} \) we have \( C, s \leadsto^{i+j} C', s', \epsilon \), as required.

**Inductive case** \( i = k+1 \) and \( k > 0 \)

Pick arbitrary \( j, C, C', C'', s, s', s'', \epsilon \) such that \( C, s \leadsto^j C'', s'', \text{ok} \) and \( C', s', \text{ok} \) and the definition of \( \leadsto^j \) we know there exists \( C''', s''' \) such that \( C, s \rightarrow C''', s''', \text{ok} \) and \( C', s', \text{ok} \). Consequently, from \( C''', s'''' \leadsto^k C'', s'', \text{ok} \), \( C'', s'' \leadsto^j C', s', \epsilon \) and the inductive hypothesis we have \( C''', s'''' \leadsto^{k+j} C', s', \epsilon \). As such, from \( C, s \rightarrow C''', s'''' \leadsto^{k+j} C', s', \epsilon \). That is, \( C, s \leadsto^{i+j} C', s', \epsilon \), as required. □

**Theorem 14** (Divergence soundness). For all \( p, C \), if \( \vdash \Box \) \( p \) \( C \infty \) can be proven using the proof rules in Fig. 3, then \( \models [p] C \infty \) holds.

**Proof.** By induction on the structure of rules in Fig. 3.

**Case** DIV-LOCAL

Pick arbitrary \( p, C \) such that \( \vdash \Box \exists x. p \) local \( x \in C \infty \). Pick an arbitrary \( s \in \exists x. p \); i.e. there exists \( u, s_p \) such that \( s_p \leftrightarrow s[x \mapsto u] \) and \( s_p \in p \). From the \( [p] C \infty \) premise and the inductive hypothesis we know there exists an infinite series \( C_1, C_2, \ldots \) and \( s_1, s_2, \ldots \) such that \( C, s_p \leadsto^1 C_1, s_1, \text{ok} \leadsto^1 C_2, s_2, \text{ok} \leadsto^1 \ldots \). As such, from the definition of \( \leadsto^+ \) and Lemma 10 we have \( C; \text{end}(x, s(x)), s_p \leadsto^+ C_1; \text{end}(x, s(x)), s_1, \text{ok} \leadsto^+ C_2; \text{end}(x, s(x)), s_2, \text{ok} \leadsto^+ \ldots \). On the other
hand, from $\text{S-LOCAL}$ we then have local $x$ in $C$, $s \rightarrow C; \text{end}(x, s(x)), s_p$. Therefore, since we also have $C; \text{end}(x, s(x)), s_p \leadsto^+ C_1; \text{end}(x, s(x)), s_1$, $ok \leadsto^+ C_2; \text{end}(x, s(x)), s_2$, $ok \leadsto^+ \cdots$, from the definition of $\leadsto^+$ we have local $x$ in $C$, $s \rightarrow^+ C_1; \text{end}(x, s(x)), s_1$, $ok \rightarrow^+ C_2; \text{end}(x, s(x)), s_2$, $ok \rightarrow^+ \cdots$, as required.

Case $\text{Div-Seq1}$

Pick arbitrary $p$, $C_1$, $C_2$ such that $\left[ p \right] C_1; C_2 [\infty]$. Pick an arbitrary $s \in p$. From the $\left[ p \right] C_1 [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $C'_2, C'_3, \cdots$, and $s_2, s_3, \cdots$, such that $C_1, s \rightarrow^+ C'_2, s_2$, $ok \rightarrow^+ C'_2, s_3$, $ok \rightarrow^+ \cdots$. As such, from the definition of $\rightarrow^+$ and Lemma 10 we have $C_1; C_2, s \rightarrow^+ C'_2; C_2, s_2$, $ok \rightarrow^+ C'_3; C_2, s_3, ok \rightarrow^+ \cdots$, as required.

Case $\text{Div-Seq2}$

Pick arbitrary $p, q$, $C_1, C_2$ such that $\left[ p \right] C_1; C_2 [\infty]$. Pick an arbitrary $s \in p$. From the $\uparrow_B \left[ p \right] C_1 [\infty]$ premise and Theorem 13 we know there exists an infinite series $C'_2, C'_3, \cdots$, and $s_2, s_3, \cdots$, such that $C_2, s_q \rightarrow^+ C'_2, s_2$, $ok \rightarrow^+ C'_2, s_3$, $ok \rightarrow^+ \cdots$. As such, from the definition of $\rightarrow^+$ and Lemma 11 we have $C_1; C_2, s \rightarrow^+ C'_2, s_2$, $ok \rightarrow^+ \cdots$. Moreover, as $C'_2, s_3 \rightarrow^+ C'_3, s_4$, $ok \rightarrow^+ \cdots$, we have $C_1; C_2, s \rightarrow^+ C'_3, s_3$, $ok \rightarrow^+ C'_4, s_4$, $ok \rightarrow^+ \cdots$, as required.

Case $\text{Div-Choice}$

Pick arbitrary $p$, $C_1, C_2$ such that $\left[ p \right] C_1 + C_2 [\infty]$. Pick an arbitrary $s \in p$ and $i \in \{1, 2\}$. From the $\left[ p \right] C_1 [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $C_3, C_4, \cdots$ and $s_3, s_4, \cdots$, such that $C_i, s \rightarrow^+ C_3, s_3$, $ok \rightarrow^+ C_4, s_4$, $ok \rightarrow^+ \cdots$. Moreover, from $\text{S-Choice}$ we have $C_1 + C_2, s \rightarrow C_i, s$, $ok$. And thus we have $C_1 + C_2, s \rightarrow C_i, s$, $ok \rightarrow^+ C_3, s_3$, $ok \rightarrow^+ C_4, s_4, ok \rightarrow^+ \cdots$, as required.

Case $\text{Div-LoopUnfold}$

Pick arbitrary $p$, $C$ such that $\left[ p \right] C^* [\infty]$. Pick an arbitrary $s \in p$. From the $\left[ p \right] C; C^* [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $C_1, C_2, \cdots$ and $s_1, s_2, \cdots$, such that $C; C^*, s \rightarrow^+ C_1, s_1$, $ok \rightarrow^+ C_2, s_2$, $ok \rightarrow^+ \cdots$. Moreover, from $\text{S-Loop}$ we have $C^*, s \rightarrow C; C^*, s$, $ok$. And thus we have $C^*, s \rightarrow C; C^*, s$, $ok \rightarrow^+ C_1, s_1$, $ok \rightarrow^+ C_2, s_2$, $ok \rightarrow^+ \cdots$, as required.

Case $\text{Div-LoopNest}$

This rule can be derived as follows:

$$\frac{\left[ p \right] C [\infty]}{\left[ p \right] C; C^* [\infty]} \text{Div-Seq1}$$

$$\frac{\left[ p \right] C; C^* [\infty]}{\left[ p \right] C^* [\infty]} \text{Div-LoopUnfold}$$

and thus this rule is sound as we established the soundness of $\text{Div-Seq1}$ and $\text{Div-LoopUnfold}$ above.

Case $\text{Div-Loop}$

Pick arbitrary $p$, $C, q$ such that $\uparrow \left[ p \right] C^* [\infty]$. From $\text{S-Loop}$ we then have:

$$\forall s \in p. C^*, s \rightarrow C; C^*, s, ok$$

(1)
From the $\vdash [p] C \[ok: q\]$ premise, Theorem 13, and the $q \subseteq p$ premise we know $\forall s \in p. \exists s' \in p. n. C, s \xrightarrow{n} \neg, s', ok$ and thus from Lemma 1, $C, s \xrightarrow{n} \neg, s', ok$. That is, from the axiom of choice we know there exist $f : p \rightarrow p$ and $g : p \rightarrow \mathbb{N}$ such that:

$$\forall s \in p. C, s \xrightarrow{g(s)} \neg, f(s), ok \land f(s) \in p$$ (2)

In what follows, we show that $\forall s \in p. C, s \xrightarrow{+} C, f(s), ok$.

Pick an arbitrary $s \in p$. From (2) we have $C, s \xrightarrow{g(s)} \neg, f(s), ok$. There are now two cases to consider: i) $g(s) = 0$; or ii) $g(s) > 0$. In case (i), we then have $C = \neg, s = f(s)$. As such, from S-SEQSKIP we have $C; C^*, s \xrightarrow{+} C, f(s), ok$, and thus by definition of $\neg^1$ we have $C; C^*, s \xrightarrow{+} C, f(s), ok$.

In case (ii), from $C, s \xrightarrow{g(s)} \neg, f(s), ok$ and Lemma 9 we have $C, s \xrightarrow{g(s)} \neg, f(s), ok$. Consequently, from Lemma 10 we have $C; C^*, s \xrightarrow{g(s)} \neg, f(s), ok$. On the other hand, from S-SEQSKIP we have $\neg, f(s) \rightarrow C^*, f(s), ok$, and thus by definition of $\neg^1$ we have $\neg, f(s) \rightarrow C^*, f(s), ok$. From Lemma 12, $C; C^*, s \xrightarrow{g(s)} \neg, f(s), ok$ and $\neg, f(s) \rightarrow C^*, f(s), ok$ we know there exist $s$ such that $C; C^*, s \xrightarrow{+} C, f(s), ok$.

That is, in both cases we know there exist $s$ such that $C; C^*, s \xrightarrow{+} C, f(s), ok$. As such, from (1) and the definition of $\neg^1$ we have $C^*, s \xrightarrow{+} C^*, f(s), ok$, i.e. $C^*, s \xrightarrow{+} C^*, f(s), ok$. That is, from (2) we have:

$$\forall s \in p. C^*, s \xrightarrow{+} C^*, f(s), ok \land f(s) \in p$$ (3)

Pick an arbitrary $s \in p$. From (3) we then know $C^*, s \xrightarrow{+} C^*, f(s), ok \xrightarrow{+} C^*, f^2(s), ok \xrightarrow{+} \ldots$, as required.

Case DIV-SUBVARIANT

Pick arbitrary $p, C, q$ such that $\vdash [p(0)] C^* [\infty]$. From S-LOOP we then have:

$$\forall s \in p. C^*, s \rightarrow C; C^*, s, ok$$ (4)

From the $\forall n \in \mathbb{N} \vdash [p(n)] C \[ok: p(n+1)]$ premise and Theorem 13 we know $\forall n \in \mathbb{N}. \forall s \in p(n). \exists s' \in p(n+1). k. C, s \xrightarrow{k} \neg, s', ok$ and thus from Lemma 1, $s \xrightarrow{k} \neg, skip, s', ok$. That is, from the axiom of choice we know there exist a series of functions, $f_1, g_1, f_2, g_2, \ldots$ such that for each $i \in \mathbb{N}$, we have $f_i : p(i-1) \rightarrow p(i)$ and $g_i : p(i-1) \rightarrow \mathbb{N}$ such that:

$$\forall i \in \mathbb{N}^+. \forall s \in p(i-1). C, s \xrightarrow{g_i(s)} \neg, f_i(s), ok \land f_i(s) \in p(i)$$ (5)

In what follows, we show that $\forall i \in \mathbb{N}^+. \forall s \in p(i-1). C^*, s \xrightarrow{+} C^*, f_i(s), ok$.

Pick an arbitrary $i \in \mathbb{N}^+$ and $s \in p(i-1)$. From (5) we have $C, s \xrightarrow{g_i(s)} \neg, f_i(s), ok$. There are now two cases to consider: a) $g_i(s) = 0$; or b) $g_i(s) > 0$. In case (a), we then have $C = \neg, s = f_i(s)$. As such, from S-SEQSKIP we have $C; C^*, s \rightarrow C; f_i(s), ok$, and thus by definition of $\neg^1$ we have $C; C^*, s \xrightarrow{+} C^*, f_i(s), ok$.

In case (b), from $C, s \xrightarrow{g_i(s)} \neg, f_i(s), ok$ and Lemma 9 we have $C, s \xrightarrow{g_i(s)} \neg, f_i(s), ok$. Consequently, from Lemma 10 we have $C; C^*, s \xrightarrow{g_i(s)} \neg, f_i(s), ok$. On the other hand, from S-SEQSKIP we have $\neg, f_i(s) \rightarrow C^*, f_i(s), ok$, and thus by definition of $\neg^1$ we have $\neg, f_i(s) \rightarrow C^*, f_i(s), ok$.

From Lemma 12, $C; C^*, s \xrightarrow{g_i(s)} \neg, f_i(s), ok$ and $\neg, f_i(s) \rightarrow C^*, f_i(s), ok$ we know there exist $f$ such that $C; C^*, s \xrightarrow{+} C^*, f_i(s), ok$.

That is, in both cases we know there exist $f$ such that $C; C^*, s \xrightarrow{+} C^*, f_i(s), ok$. As such, from (4) and the definition of $\neg^1$ we have $C^*, s \xrightarrow{+} C^*, f_i(s), ok$, i.e. $C^*, s \xrightarrow{+} C^*, f_i(s), ok$. That is,
from (5) we have:
\[
\forall i \in \mathbb{N}^+. \forall s \in p(i-1). \ C^*, s \leadsto^+ C^*, f_i(s), \ ok \land f_i(s) \in p(i)
\] (6)

Pick an arbitrary \(s \in p(0)\). From (6) we then know \(C^*, s \leadsto^+ C^*, f_1(s), \ ok \leadsto^+ C^*, f_2(s), \ ok \leadsto^+ \ldots\), as required.

**Case Div-Cons**

Pick arbitrary \(p, C\) such that \(\vdash [p] C \[\omega\]\). Pick an arbitrary \(s \in p\). From the \(p \subseteq p'\) premise we know \(s \in p'\). As such, from the \([p'] C \[\omega\]\) premise we know there exists an infinite series \(C_1, C_2, \ldots\) and \(s_1, s_2, \ldots\), such that \(C, s \leadsto^+ C_1, s_1, \ ok \leadsto^+ C_2, s_2, \ ok \leadsto^+ \ldots\), as required.

**Case Div-Subst**

Pick arbitrary \(p, C, q, y\) such that \(y \notin \text{fv}(p, C)\) and \(\vdash [p] C \[\omega\])[y/x]\), i.e. \(\vdash [p[y/x]] C[y/x] \[\omega\]\). Pick an arbitrary \(s \in p[y/x]\) and let \(s_p = s[x \mapsto s(y)]\). We then have \(s_p \in p\) and thus from the \(\vdash_B [p] C \[\epsilon : q\]\) premise and the inductive hypothesis we then know there exists an infinite series \(C_1, C_2, \ldots\) and \(s_1, s_2, \ldots\) such that \(C, s_p \leadsto^+ C_1, s_1, \ ok \leadsto^+ C_2, s_2, \ ok \leadsto^+ \ldots\). Let \(C'_i = C_i[y/x]\) and \(s'_i = s_i[y \mapsto s'_i(y)]\) for all \(i\) As such, from the semantics we also have \(C[y/x], s \leadsto^+ C'_1, s'_1, \ ok \leadsto^+ C'_2, s'_2, \ ok \leadsto^+ \ldots\), as required. \(\square\)
C UNTER COMPLETENESS

C.1 Completeness of BU and FUA Rules

Lemma 13 (BUA completeness). For all \( p, q, \epsilon \), if \( \models_B [p] C [\epsilon : q] \) holds, then \( \vdash_B [p] C [\epsilon : q] \) can be proven using the proof rules in Fig. 2.

Proof. We proceed by induction of the structure of \( C \).

Case: \( C = \text{skip} \)

Pick arbitrary \( p, q \) such that \( \models_B [p] \text{skip} [\epsilon : q] \) holds. Given the semantics of skip, we then know \( p \subseteq q \). As such, we can derive \( \vdash_B [p] C [\epsilon : q] \) using SKIP and CONS\( F \).

Cases: \( C = \text{assume}(B) \) and \( C = \text{error} \)

The proofs of these cases are analogous to the \( C = \text{skip} \) case and omitted.

Case: \( C = x := \epsilon \)

Pick arbitrary \( p, q \) such that \( \models_B [p] x := e [\epsilon : q] \) holds. As \( \exists y. p[y/x] \land x = e[y/x] \) is the strongest post of \( x := e \) from \( p \) (see [23]), we then know \( \exists y. p[y/x] \land x = e[y/x] \subseteq q \). Moreover, from ASSIGN we have \( \vdash_B [p] x := e [\epsilon : \exists y. p[y/x] \land x = e[y/x]] \). Consequently, as \( \exists y. p[y/x] \land x = e[y/x] \subseteq q \) from CONS\( F \), we have \( \vdash_B [p] x := e [\epsilon : q] \), as required.

Case: \( C = \text{local} \ x \ in \ C \)

Pick arbitrary \( p, q \) such that \( \models_B [p] \text{local} \ x \ in \ C [\epsilon : q] \) holds. Pick an arbitrary \( y \) such that \( y \notin \text{fv}(C) \), \( y \notin \text{fv}(p) \) and \( y \notin \text{fv}(q) \). Then we know that local \( y \) in \( C \) is semantically equivalent to local \( x \) in \( C \) and thus \( \models_B [p] \text{local} \ y \ in \ C [\epsilon : q] \) holds. From the semantics of local \( y \) in \( C \) we know there exist \( v_1, v_2 \) such that \( \models_B [p \land y = v_1] C [\epsilon : q \land y = v_2] \) holds. From the inductive hypothesis we then have \( \vdash_B [p \land y = v_1] C [\epsilon : q \land y = v_2] \), and thus from LOCAL we have \( \vdash_B [\exists y. p \land y = v_1] \text{local} \ y \ in \ C \ [\epsilon : q \land \exists y. y = v_2] \). As \( y \notin \text{fv}(p) \) and \( y \notin \text{fv}(q) \), using CONS\( E_{\text{q}} \) we have \( \vdash_B [p \land \exists y. y = v_1] \text{local} \ y \ in \ C \ [\epsilon : q \land \exists y. y = v_2] \). Finally, using SUBST and since \( y \notin \text{fv}(C) \), \( y \notin \text{fv}(p) \) and \( y \notin \text{fv}(q) \), we have \( \vdash_B [p] \text{local} \ x \ in \ C [\epsilon : q] \), as required.

Case: \( C = C_1; C_2 \)

Pick arbitrary \( p, q \) such that \( \models_B [p] C_1; C_2 [\epsilon : q] \) holds. From the semantics of \( C_1; C_2 \) we then know either 1) \( \epsilon = \text{er} \) and \( \models_B [p] C_1 [\text{er} : q] \); or 2) \( \epsilon = \text{ok} \) and there exists \( r \) such that \( \models_B [p] C_1 [\text{ok} : r] \) and \( \models_B [r] C_2 [\epsilon : q] \). In case (1) from \( \models_B [p] C_1 [\text{er} : q] \) and the inductive hypothesis we know we can prove \( \vdash_B [p] C_1 [\text{er} : q] \), and thus using SEQ\( E \) we can prove \( \vdash_B [p] C_1; C_2 [\epsilon : q] \), as required. In case (2) from \( \models_B [p] C_1 [\text{ok} : r] \) and \( \models_B [r] C_2 [\epsilon : q] \) and the inductive hypotheses we know we can prove \( \vdash_B [p] C_1 [\text{ok} : r] \) and \( \vdash_B [r] C_2 [\epsilon : q] \). Consequently, using SEQ we can prove \( \vdash_B [p] C_1; C_2 [\epsilon : q] \), as required.

Case: \( C = C_1 + C_2 \)

Pick arbitrary \( p, q \) such that \( \models_B [p] C_1 + C_2 [\epsilon : q] \) holds. From the semantics of \( C_1 + C_2 \) we know there exists \( i \in \{1, 2\} \) such that \( \models_B [p] C_i [\epsilon : q] \). From \( \models_B [p] C_i [\epsilon : q] \) and the inductive hypothesis we know we can prove \( \vdash_B [p] C_i [\epsilon : q] \), and thus using CHOICE we can prove \( \vdash_B [p] C_1 + C_2 [\epsilon : q] \), as required.

Case: \( C = C^* \)

Pick arbitrary \( p, q \) such that \( \models_B [p] C^* [\epsilon : q] \) holds. There are two cases to consider: 1) \( \epsilon = \text{ok} \); or
$e = er$. In case (1), let $p(0) = p$ and $p(n)$ be the state reachable after executing $C$ $n$ times starting from $p(0)$ for $n > 0$. By definition we then know there exists $k > 0$ such that $q = p(k)$. Moreover, by definition we then have $\vdash_B [p(n)] C [ok: p(n+1)]$ for all $0 \leq nk$. As such, by the inductive hypothesis we have $\vdash_B [p(n)] C [ok: p(n+1)]$ for all $n < k$. Using \textsc{loop-subvariant} we then have $\vdash_B [p(0)] C [ok: p(k)]$, i.e. $\vdash_B [p] C [ok: q]$, as required.

In case (2), from the semantics of loops we know that $C$ executed normally for a number of (possibly zero) iterations, and in the subsequent iteration the loop encountered an error. That is, there exist $r$ such that $\vdash_B [p] C^* [ok: r]$ and $\vdash_B [r] C [er: q]$. From the proof of case (1) we then have $\vdash_B [p] C^* [ok: r]$. From $\vdash_B [r] C [er: q]$ and the inductive hypothesis we have $\vdash_B [r] C [er: q]$. Consequently, from $\vdash_B [p] C^* [ok: r], \vdash_B [r] C [er: q]$ and \textsc{seq} we have $\vdash_B [p] C^*; C [er: q]$, i.e. $\vdash_B [p] C^*; C [\epsilon: q]$. As such, from \textsc{Loop} we have $\vdash_B [p] C^* [er: q]$, as required. □

Lemma 14 (FUA completeness). For all $p, C, q, e$, if $\models_F [p] C [e: q]$ holds, then $\vdash_F [p] C [e: q]$ can be proven using the proof rules in Fig. 2.

Proof. The proof of this lemma is as given by O’Hearn [23].

Theorem 15 (Completeness). For all $p, C, q, e$, if $\models_T [p] C [e: q]$ holds, then $\vdash_T [p] C [e: q]$ can be proven using the proof rules in Fig. 2.

Proof. Follows immediately from Lemma 13 and Lemma 14. □

C.2 Completeness of Divergence Rules

In what follows, we write $C, s \leadsto^+ C', s', e$ for $\exists n. C, s \leadsto^n C', s', e$.

\[
\text{Div-Subst} \quad \vdash [p] C [\infty] \quad y \notin fv(p, C) \\
\vdash ([p] C [\infty])[y/x]
\]

Theorem 16 (Divergence completeness). For all $p, C$, if $\models [p] C [\infty]$ holds, then $\vdash [p] C [\infty]$ can be proven using the proof rules in Fig. 3.

Proof. We proceed by induction of the structure of $C$.

Cases $C = \text{skip}$, $C = x := e$, $C = \text{error}$, $C = \text{assume}(B)$.

These cases do not arise as they have no divergent steps and reduce to skip in either 0 or 1 steps.

Case $C = \text{local } x \in C$

Pick arbitrary $p$ such that $\models [p] \text{local } x \in C [\infty]$ holds. Pick an arbitrary $y$ such that $y \notin fv(C)$ and $y \notin fv(p)$. Then we know that local $y \in C$ is semantically equivalent to local $x \in C$ and thus $\models [p] \text{local } y \in C [\infty]$ holds. From the semantics of local $y \in C$ we know there exist $v_1$ such that $\models [p \land y = v_1] C [\infty]$ holds. From the inductive hypothesis we then have $\vdash [p \land y = v_1] C [\infty]$, and thus from \textsc{div-local} we have $\vdash [\exists y. p \land y = v_1] \text{local } y \in C [\infty]$. As $y \notin fv(p)$, using \textsc{div-cons} we have $\vdash [p \land \exists y. y = v_1] \text{local } y \in C [\infty]$. Once again, using \textsc{div-cons} we have $\vdash [p] \text{local } y \in C [\infty]$. Finally, using \textsc{div-subst} and since $y \notin fv(C)$ and $y \notin fv(p)$, we have $\vdash [p] \text{local } x \in C [\infty]$, as required.

Case $C = C_1; C_2$

Pick arbitrary $p$ such that $\models [p] C_1; C_2 [\infty]$ holds. From the semantics of $C_1; C_2$ we then know either 1) $\models [p] C_1 [\infty]$; or 2) there exists $q$ such that $\models_B [p] C_1 [ok: q]$ and $\models [q] C_2 [\infty]$. In case (1) from $\models [p] C_1 [\infty]$ and the inductive hypothesis we know we can prove $\vdash [p] C_1 [\infty]$, and thus
using \textsc{Div-Seq1} we can prove $\vdash [p] C_1; C_2 [\infty]$, as required. In case (2) from $\models_B [p] C_1 [\ok: q]$ and Theorem 15 we have $\vdash_B [p] C_1 [\ok: q]$. Moreover, from $\models [q] C_2 [\infty]$ and the inductive hypotheses we can prove $\vdash [q] C_2 [\infty]$. Consequently, using \textsc{Div-Seq2} we can prove $\vdash [p] C_1; C_2 [\infty]$, as required.

**Case $C = C_1 + C_2$**

Pick arbitrary $p$ such that $\models [p] C_1 + C_2 [\infty]$ holds. From the semantics of $C_1 + C_2$ we know there exists $i \in \{1, 2\}$ such that $\models [p] C_i [\infty]$ holds. From $\models [p] C_i [\infty]$ and the inductive hypothesis we know we can prove $\vdash [p] C_i [\infty]$, and thus using \textsc{Div-Choice} we can prove $\vdash [p] C_1 + C_2 [\infty]$, as required.

**Case $C = C^*$**

Pick arbitrary $p$ such that $\models [p] C^* [\infty]$ holds. Let $p(0) = p$ and $p(n)$ be the state reachable after executing $C$ for $n$ times starting from $p(0)$ for $n > 0$. Let $C^0 = \text{skip}$ and let $C^n$ denote iterating $C$ for $n$ times, for all $n > 0$. Given the semantics of loops, there are two cases to consider: There are two cases to consider:

1) $\models_B [p(n)] C [\ok: p(n+1)]$ for all $n \in \mathbb{N}$; or

2) there exists $n$ and $q$ such that $\models [p] C^n [\ok: q]$ and $\models [q] C [\infty]$.

In case (1), from Theorem 15 we have $\vdash_B [p(n)] C [\ok: p(n+1)]$ for all $n \in \mathbb{N}$. As such, using \textsc{Div-Subvariant} we have $\vdash [p(0)] C [\infty]$, i.e. $\vdash [p] C [\infty]$, as required.

In case (2), we proceed by induction on $n$.

**Subcase $n = 0$**

As we have $\models_B [p] C^n [\epsilon : q], \models [q] C [\infty]$ and $C^0 = \text{skip}$, we know $p \subseteq q$. Moreover, from $\models [q] C [\infty]$ and the inductive hypothesis we have $\vdash [q] C [\infty]$, and as such from \textsc{Div-LoopNest} we have $\vdash [q] C^* [\infty]$. Consequently, as $p \subseteq q$, from \textsc{Div-Cons} we have $\vdash [p] C^* [\infty]$, as required.

**Subcase $n = k+1$**

From $\models_B [p] C^n [\ok: q]$ and Theorem 15 we have

$\vdash_B [p] C^n [\ok: q]$  \hspace{1cm} (7)

Moreover, from $\models [q] C [\infty]$ and the inductive hypothesis we have

$\vdash [q] C [\infty]$  \hspace{1cm} (8)

As $C^n = C; \cdots; C$, we can then prove $\vdash [p] C^* [\infty]$ as follows:

\hspace{2cm}$\vdash_B [p] C^n [\ok: q]$ \hspace{1cm} \textsc{Div-LoopNest}

\hspace{2cm}$\vdash [q] C [\infty]$ \hspace{1cm} \textsc{Div-Seq2}

\hspace{2cm}$\vdash [p] C^n; C^* [\infty]$ \hspace{1cm} \textsc{Div-LoopUnfold} \times n

\hspace{2cm}$\vdash [p] C^* [\infty]$ \hspace{1cm} \textsc{Div-LoopUnfold} \times n

$\square$
THE RELATION BETWEEN FUA AND BUA TRIPLES

**Theorem 17.** For all $p, C, q, \epsilon$, if $\models_F [p] C [\epsilon : q]$ holds and $\min_{\text{pre}}(p, C, q)$ also holds, where

$$\min_{\text{pre}}(p, C, q) \iff \forall p'. p' \subset p \Rightarrow \not\models_F [p'] C [\epsilon : q]$$

**Proof.** Pick arbitrary $p, C, q, \epsilon$ such that $\models_F [p] C [\epsilon : q]$ holds and $\min_{\text{pre}}(p, C, q)$. Let us proceed by contradiction and assume that $\models_B [p] C [\epsilon : q]$ does not hold. That is, there exists $s_p \in p$ such that:

$$\not\exists s_q \in q, n. C, s_p \overset{n}{\sim} s_q, \epsilon$$  \hfill (9)

Let $p' \doteq p \setminus \{s_p\}$. We next show that $\models_F [p'] C [\epsilon : q]$ holds. Pick an arbitrary $s_2 \in q$. Since $\models_F [p] C [\epsilon : q]$ holds, from its definition we know there exists $s_1 \in p$, $k$ such that $C, s_1 \overset{k}{\rightarrow} s_2, \epsilon$. However, from (9) we know $s_1 \neq s_p$. Consequently, since $p' \doteq p \setminus \{s_p\}$ and $s_1 \in p$, we know $s_1 \in p'$. That is, there exists $s_1 \in p', k$ such that $C, s_1 \overset{k}{\rightarrow} s_2, \epsilon$, and thus by definition we have:

$$\models_F [p'] C [\epsilon : q]$$  \hfill (10)

Finally, from $\min_{\text{pre}}(p, C, q)$, (10) and the definition of $\min_{\text{pre}}$ we have $p' \not\subset p$. This, however, leads to a contradiction as $p' \neq p \setminus \{s_p\}$ and thus $p' \subset p$. \hfill \Box

**Theorem 18.** For all $p, C, q, \epsilon$, if $\models_B [p] C [\epsilon : q]$ holds and $\min_{\text{post}}(p, C, q)$ also holds, where

$$\min_{\text{post}}(p, C, q) \iff \forall q'. q' \subset q \Rightarrow \not\models_B [p] C [\epsilon : q']$$

**Proof.** Pick arbitrary $p, C, q, \epsilon$ such that $\models_B [p] C [\epsilon : q]$ holds and $\min_{\text{post}}(p, C, q)$. Let us proceed by contradiction and assume that $\models_F [p] C [\epsilon : q]$ does not hold. That is, there exists $s_q \in q$ such that:

$$\not\exists s_p \in p, n. C, s_p \overset{n}{\sim} s_q, \epsilon$$  \hfill (11)

Let $q' \doteq q \setminus \{s_q\}$. We next show that $\models_B [p] C [\epsilon : q']$ holds. Pick an arbitrary $s_1 \in p$. Since $\models_B [p] C [\epsilon : q]$ holds, from its definition we know there exists $s_2 \in q$, $k$ such that $C, s_1 \overset{k}{\rightarrow} s_2, \epsilon$. However, from (11) we know $s_2 \neq s_q$. Consequently, since $q' \doteq q \setminus \{s_q\}$ and $s_2 \in q$, we know $s_q \in q'$. That is, there exists $s_2 \in q', k$ such that $C, s_1 \overset{k}{\rightarrow} s_2, \epsilon$, and thus by definition we have:

$$\models_B [p] C [\epsilon : q']$$  \hfill (12)

Finally, from $\min_{\text{post}}(p, C, q)$, (12) and the definition of $\min_{\text{post}}$ we have $q' \not\subset q$. This, however, leads to a contradiction as $q' \neq q \setminus \{s_q\}$ and thus $q' \subset q$. \hfill \Box
E  UNTerSL MODEL AND SEMANTICS

Separation Logic at a Glance. The essence of SL and its compositional reasoning power is embodied in its frame rule, adapted to our notation below (left), which enables one to extend the underlying heap (memory) arbitrarily with additional resources (described by \( r \)), allowing the same specification (triple) to be reused in different contexts with different heaps. Intuitively, the heaps described by the frame \( r \) lie outside the footprint of \( C \) (parts of the heap accessed and modified by \( C \)), as stipulated by \( \text{mod}(C) \cap \text{fv}(r) = \emptyset \), and thus this frame remains unchanged when executing \( C \). The \( \ast \) connective denotes the separating conjunction (read as ‘and separately’), and is used to combine the underlying heaps (by taking their union provided that they contain distinct locations).

\[
\text{SL-Frame} \quad \frac{C[e : q]}{\vdash F [p \ast r]} \quad \frac{\text{mod}(C) \cap \text{fv}(r) = \emptyset}{\vdash F [p \ast r]} \quad C[e : q \ast r]
\]

The compositionality afforded by \text{SL-Frame} allows us to devise \textit{local} axioms describing the behaviour of heap-manipulating operations, in that we can only mention those parts of the heap manipulated by the operation and later extend this behaviour to larger (global) settings by using \text{SL-Frame}. For instance, we can describe the behaviour of freeing memory as in the \text{SL-Free} axiom above (middle), adapted from the corresponding SL axiom. Specifically, the \( x \mapsto v \) assertion describes a state in which the heap comprises a single location at \( x \) holding value \( v \). Moreover, \( x \mapsto v \) describes a (linear) resource that grants exclusive permission to access location \( x \) and thus cannot be duplicated; i.e. for all \( x, v \) and \( v' : x \mapsto v \ast x \mapsto v' \Leftrightarrow \text{false} \). On the other hand, the \( \text{emp} \) assertion describes states in which the heap is empty, and thus represents the identity of \( \ast \)-composition; i.e. for all \( p : p \ast \text{emp} \Leftrightarrow p \).

FUA Triples and Separation Logic. To achieve compositional reasoning, an SL extension of a FUA reasoning system must preserve the soundness of \text{SL-Frame}. However, as Raad et al. [24] show, the original model of SL is unsound for FUA reasoning. In particular, we can apply \text{SL-Frame} with \( r \triangleq x \mapsto v \) as shown below, resulting in an invalid FUA triple:

\[
\vdash F [x \mapsto v] \vdash F [\text{free}(x) \mid \text{ok} : \text{emp}] \\
\vdash F [x \mapsto v \ast x \mapsto v] \vdash F [\text{free}(x) \mid \text{ok : emp} \ast x \mapsto v] \\
\vdash F [\text{false}] \vdash F [\text{free}(x) \mid \text{ok} : x \mapsto v]
\]

Note that [\text{false}] free(x) [\text{ok} : x \mapsto v] in the conclusion is unsound: it states that every state in x \mapsto v can be reached from some state in false, while false denotes an empty set of states.

To remedy this, Raad et al. [24] propose an adapted model in which they track the knowledge that a location was previously freed via negative heap assertions. Specifically, a negative heap assertion, \( x \not\mapsto \), conveys: 1) the knowledge that \( x \) is an addressable location; 2) the knowledge that \( x \) is not allocated; and 3) the ownership of location \( x \). That is, \( x \not\mapsto \) is analogous to the points-to assertion \( x \mapsto - \) and is thus manipulated similarly using \( \ast \)-conjunction. More concretely, one cannot consistently \( \ast \)-conjoin \( x \not\mapsto \) either with \( x \mapsto - \) or with itself: \( x \mapsto - \ast x \not\mapsto \Leftrightarrow \text{false} \) and \( x \not\mapsto \ast x \not\mapsto \Leftrightarrow \text{false} \). Using negative assertions, one can specify the free(x) axiom as in \text{ISL-Free} above (right), recovering the frame rule: this time, when we frame \( x \mapsto v \) on both sides, we obtain the inconsistent assertion \( x \mapsto - \ast x \not\mapsto \) on the right-hand side (i.e. we have false as both pre- and post-states), which always renders a FUA triple vacuously valid.

Assertion Semantics. We present the semantics of UNTerSL assertions at the top of Fig. 10, where an assertion is interpreted as a set of UNTerSL states. The semantics of classical assertions
**Small-Step Operational Semantics.** We present the UNTER\textsuperscript{SL} operational semantic in Fig. 8 (below). As seen in SL-LOCAL–SL-LOOP, the UNTER\textsuperscript{SL} semantics of constructs imported from UNTER are analogous to their UNTER counterparts and are simply lifted to operate on UNTER\textsuperscript{SL} states.

The remaining transitions pertain to heap-manipulating operations. Specifically, SL-ALLOC describes executing \( x := \text{alloc}() \), where a previously unallocated location \( l \) is picked, the underlying heap is extended with \( l \), and \( x \) is updated in the store to record \( l \). Similarly, SL-ALLOCFREE describes re-allocating a location \( l \) that was previously deallocated (i.e. \( h(l) = \bot \)). The SL-FREE transition describes successfully dealocating the memory at \( x \): when \( x \) holds location \( l \) (\( s(x) = l \)) and \( l \) is allocated in the memory (\( h(l) \in \text{VAL} \)), then \( l \) is deallocated by updating its value to \( \bot \) in the heap. Conversely, SL-FREE describes when deallocating the memory at \( x \) fails, namely when either \( x \) holds null or \( x \) holds a location that has already been deallocated, in which case the underlying state is unchanged. Analogously, SL-LOAD and SL-LOADER respectively describe reading from memory via \( x := [y] \) successfully (when \( y \) holds an allocated location) and erroneously (when \( y \) holds either null or a deallocated location). Finally, SL-STORE and SL-STOREER respectively describe writing to memory successfully and erroneously.
\( \mathcal{L} : \text{AST} \rightarrow \mathcal{P}(\text{STaTE}^{\text{SL}}) \)

\[
\begin{align*}
\langle \text{emp} \rangle & \triangleq \{(s, h) \mid \text{dom}(h) = \emptyset\} \\
\langle e \mapsto e' \rangle & \triangleq \{(s, h) \mid \text{dom}(h) = \{s(e)\} \land h(s(e)) = s(e') \neq \perp\} \\
\langle e \not\mapsto \rangle & \triangleq \{(s, h) \mid \text{dom}(h) = \{s(e)\} \land h(s(e)) = \perp\} \\
\langle p \cdot q \rangle & \triangleq \{s \cup \sigma_p \cdot \sigma_q \mid s \in \langle p \rangle \land \sigma_q \in \langle q \rangle\}
\end{align*}
\]

where \((s, h) \circ (s', h') \triangleq \begin{cases} 
(s, h \uplus h') & \text{if } s = s' \land \text{dom}(h(1)) \cap \text{dom}(h(2)) = \emptyset \land \text{wf}(h \uplus h') \\
\text{undefined} & \text{otherwise}
\end{cases}\)

---

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL-LOCAL</td>
<td>( s' = s[x \mapsto v] \quad v \in \text{VAL} )</td>
</tr>
<tr>
<td>SL-LOCALEnd</td>
<td>( \text{end}(x, v), (s, h) \rightarrow \text{skip}, (s', h) )</td>
</tr>
<tr>
<td>SL-ASSUME</td>
<td>( s(B) = \text{true} )</td>
</tr>
<tr>
<td>SL-ASSIGN</td>
<td>( x := e, (s, h) \rightarrow \text{skip}, (s', h), \text{ok} )</td>
</tr>
<tr>
<td>SL-ERROR</td>
<td>( \text{error}, \sigma \rightarrow \sigma, \text{er} )</td>
</tr>
<tr>
<td>SL-CHOICE</td>
<td>( i \in {1, 2} )</td>
</tr>
<tr>
<td>SL-SEQ0</td>
<td>( C_1 + C_2, \sigma \rightarrow C_1, \sigma, \text{ok} )</td>
</tr>
<tr>
<td>SL-SEQ1</td>
<td>( C_1; C_2, \sigma \rightarrow C_1'; \sigma \rightarrow C_2; \sigma', \epsilon )</td>
</tr>
<tr>
<td>SL-SEQSkip</td>
<td>( \text{skip}; C, \sigma \rightarrow C; \sigma, \text{ok} )</td>
</tr>
<tr>
<td>SL-LOOP0</td>
<td>( C^*; \sigma \rightarrow \sigma, \text{ok} )</td>
</tr>
<tr>
<td>SL-LOOP</td>
<td>( C^*; \sigma \rightarrow C; \sigma, \text{ok} )</td>
</tr>
<tr>
<td>SL-ALLOCFree</td>
<td>( h(l) = \perp )</td>
</tr>
<tr>
<td>SL-LOAD</td>
<td>( s(x) = \text{null} \lor h(s(x)) = \perp )</td>
</tr>
<tr>
<td>SL-LOAD</td>
<td>( h(y) = v \in \text{VAL} )</td>
</tr>
<tr>
<td>SL-STORE</td>
<td>( s(y) = l )</td>
</tr>
<tr>
<td>SL-STORE</td>
<td>( s(x) \rightarrow e, \sigma \rightarrow \sigma, \text{er} )</td>
</tr>
<tr>
<td>SL-STORE</td>
<td>( x \rightarrow [y], (s, h) \rightarrow \text{skip}, (s, h), \text{er} )</td>
</tr>
<tr>
<td>SL-STORE</td>
<td>( x \rightarrow y, (s, h) \rightarrow \text{skip}, (s, h), \text{ok} )</td>
</tr>
</tbody>
</table>

---

Fig. 10. The semantics of \( \text{UNTER}^{\text{SL}} \) assertions (above); the \( \text{UNTER}^{\text{SL}} \) small-step operational semantics (below)
\section{\textsc{Unter}袜 Soundness}

\begin{definition}
\label{def:untersoundness}
s_1 \sim_A s_2 \iff \forall x \in A. \ s_1(x) = s_2(x)
\end{definition}

\begin{definition}
\label{def:compatibility}
h_p \# h \iff \text{dom}(h_p) \cap \text{dom}(h) = \emptyset \\
\sigma_p \# \sigma \iff \exists \sigma'. \ \sigma_p \circ \sigma = \sigma'
\end{definition}

Intuitively, \(h_p \# h\) (resp. \(\sigma_p \# \sigma\)) denotes that \(h_p\) and \(h\) (resp. \(\sigma_p\) and \(\sigma\)) are compatible in that their composition is defined.

\begin{proposition}
\label{prop:untersoundness}
For all assertions \(p\) and all \(s, s', h\), if \((s, h) \in p\) and \(s \sim_{\text{fv}(p)} s'\), then \((s', h) \in p\).
\end{proposition}

For all \(\epsilon, C, x, v, n, (s_1, h_1)\) and \((s_2, h_2)\), if \(C, (s_1, h_1) \rightarrow \sim, (s_2, h_2), \epsilon\) and \(x \notin \text{fv}(C)\), then \(C, ((s_1[x \mapsto v], h_1) \sim, (s_2[x \mapsto v], h_2)), \epsilon\).

\subsection{BUA Soundness in \textsc{Unter}袜}

\begin{lemma}
\label{lemma:untersoundness}
For all \(n, \sigma, \sigma', C, C', \) if \(C, \sigma \rightarrow^n C', \sigma', \) then \(C' = \text{skip}\).
\end{lemma}

\begin{proof}
The proof of this lemma is analogous to that of \text{Lemma 1} and is omitted here. \hfill \Box
\end{proof}

\begin{lemma}
\label{lemma:untersoundness2}
For all \(p, C\):
\begin{align}
\begin{aligned}
\forall n \in \mathbb{N}, (s, h_p) \in p(n), h. \ h_p \# h & \Rightarrow \exists (s', h_q) \in p(n+1), j. \ s \sim_{C^*} s' \land C, (s, h_p \uplus h) \rightarrow^j \\
& \sim, (s', h_q \uplus h), \text{ok}.
\end{aligned}
\end{align}
\end{lemma}

\begin{proof}
Pick arbitrary \(i, C\) such that:
\begin{align}
\forall n \in \mathbb{N}, (s, h_p) \in p(n), h. \ h_p \# h & \Rightarrow \exists (s', h_q) \in p(n+1), j. \ s \sim_{C^*} s' \land C, (s, h_p \uplus h) \rightarrow^j \\
& \sim, (s', h_q \uplus h), \text{ok}
\end{align}
\end{proof}

We proceed by induction on \(k\).

\begin{basecase}
\(k = 0\)
Pick an arbitrary \(i, (s, h_p) \in p(i)\) and \(h\) such that \(h_p \# h\). We then simply have \(s \sim_{C^*} s\).
\end{basecase}

From \text{S-Loop\(0\)} we have \(C^*, (s, h_p \uplus h) \rightarrow \text{skip}, (s, h_p \uplus h), \text{ok}\). As such, as we have \(s \sim_{C^*} s\), \(\text{skip}, (s, h_p \uplus h), \text{ok}\) (from the definition of \(\rightarrow\)), by definition we have \(C^*, (s, h_p \uplus h) \rightarrow \text{skip}, (s, h_p \uplus h), \text{ok}\). Consequently, we have \((s, h_p) \in p(i)\) and \(C^*, (s, h_p \uplus h) \rightarrow \text{skip}, (s, h_p \uplus h), \text{ok}\), as required.

\begin{inductivecase}
\(k = j+1\)
\begin{align}
\forall i \in \mathbb{N}, (s, h_p) \in p(i), h. \ h_p \# h & \Rightarrow \exists (s', h_q) \in p(i+j), m. \ s \sim_{C^*} s' \land C^*, (s, h_p \uplus h) \rightarrow^m \sim, (s', h_q \uplus h), \text{ok}
\end{align}
\end{inductivecase}

Pick an arbitrary \(i \in \mathbb{N}, (s, h_p) \in p(i)\) and \(h\) such that \(h_p \# h\). From \text{Lemma 13} and since \((s, h_p) \in p(i)\) we know there exists \((s_i, h_i) \in p(i+1)\) and \(m\) such that \(s \sim_{C^*} s_i \land C, (s, h_p \uplus h) \rightarrow^m \sim, (s_i, h_i \uplus h), \text{ok}\).

That is, \(h_i \# h\). As \(s \sim_{C^*} s_i \land C \rightarrow \text{skip}, (s_i, h_i \uplus h), \text{ok}\), we also have \(s \sim_{C^*} s_i\).

On the other hand, since \((s_i, h_i) \in p(i+1)\) and \(h_i \# h\), from \text{I.H} we know there exists \((s', h_q) \in p(i+1+j)\) and \(b\) such that \(s_i \sim_{C^*} s', (s_i, h_i \uplus h) \rightarrow^b \sim, (s', h_q \uplus h), \text{ok}\). That is, \((s', h_q) \in p(i+k)\).
Therefore, from Lemma 2, $C, (s, h_p \uplus h) \xrightarrow{m} (s_i, h_i \uplus h), ok$ and $C^*, (s_i, h_i \uplus h) \xrightarrow{b} (s', h_q \uplus h), ok$.

we know there exists $c$ such that $C; C^*, (s, h_p \uplus h), \sim (s', h_q \uplus h), ok$.

Furthermore, from S-Loop we simply have $C^*, (s, h_p \uplus h), \rightarrow C; C^*, (s, h_p \uplus h), ok$. As such, since we also have $C; C^*, (s, h_p \uplus h), \sim (s', h_q \uplus h), ok$, from the definition of $C^+$ we have $C^*, (s, h_p \uplus h), \xrightarrow{c+1} (s', h_q \uplus h), ok$. Finally, since $s \xrightarrow{\text{mod}(C^+)} s_i$ and $s_i \xrightarrow{\text{mod}(C^+)} s'$, we also have $s \xrightarrow{\text{mod}(C^+)} s'$. That is, we have $(s', h_q) \in p(i+k), s \xrightarrow{\text{mod}(C^+)} s' \land C^*, (s, h_p \uplus h), \xrightarrow{c+1} (s', h_q \uplus h), ok$, as required. 

\[\text{Lemma 17. For all } p, C, q, \epsilon, \text{ if } T_B \vdash [p] C [e : q] \text{ can be derived using the proof rules in Fig. 9, then:}\]

\[
\exists(s_q, h_q) \in q, n. \ s \xrightarrow{\text{mod}(C^+)} s_q \land C, (s_p, h_p \uplus h) \xrightarrow{n} (s_q, h_q \uplus h), \epsilon
\]

\[\text{PROOF. By induction on the structure of rules in Fig. 9.}\]

\[\text{Case Skip}\]

Pick an arbitrary $\sigma_p = (s, h_p) \in p$ and an arbitrary $h$ such that $h_p \neq h$. It then suffices to show that skip, $(s, h_p \uplus h) \xrightarrow{0} \text{skip, (s, h_p \uplus h), ok}$, which follows from the definition of $\xrightarrow{0}$ immediately.

\[\text{Case Assign}\]

Pick an arbitrary $\sigma_p \in x = x'$ and an arbitrary $h$ such that $h_p \neq h$. That is, there exists $s$ such that $\sigma_p = (s, \emptyset)$. Let $s(x) = v_x, s(e) = v_e$ and $s' = s[x \mapsto v_e]$. As $\sigma_p = (s, \emptyset) \in x = x'$ we also have $s(x') = v_x$. As $\text{mod}(x := e) = \{x\}$, by definition of $s'$ we have $s \xrightarrow{\text{mod}(x := e)} s'$. From S-Assign we then have $x := e, (s, h) \rightarrow \text{skip, (s', h), ok}$. As such, since we also have skip, $(s', h) \xrightarrow{0} \text{skip, (s', h), ok}$, by definition we have $x := e, (s, h) \xrightarrow{1} \text{skip, (s', h), ok}$, i.e. $x := e, (s, \emptyset \uplus h) \xrightarrow{1} \text{skip, (s', \emptyset \uplus h), ok}$.

As $s(x) = s(x') = v_x$ and $s(e) = v_e$, by definition we have $s(e[x'/x]) = v_e$ and $s'(e[x'/x]) = v_e$. As $s'(e[x'/x]) = v_e$ and $s' = s[x \mapsto v_e]$ (i.e. $s'(x) = v_e$), we have $(s', \emptyset) \in x = e[x'/x]$. Therefore, we have $(s', \emptyset) \in x = e[x'/x], s \xrightarrow{\text{mod}(x := e)} s'$ and $x := e, (s, \emptyset \uplus h) \xrightarrow{1} \text{skip, (s', \emptyset \uplus h), ok}$, as required.

\[\text{Case Assume}\]

Pick arbitrary $p, B$ such that $T_B \vdash [p \land B] \text{ assume}(B) [ok: p \land B]$. Pick an arbitrary $(s, h_p) \in p \land B$ and an arbitrary $h$ such that $h_p \neq h$. By definition we then know $s(B) = \text{true}$. As $\text{mod} (\text{assume}(B)) = \emptyset$, by definition we have $s \xrightarrow{\text{mod}(\text{assume}(B))} s$. From S-Assume we then have $\text{assume}(B), (s, h_p \uplus h) \rightarrow \text{skip, (s, h_p \uplus h), ok}$. As such, since we also have skip, $(s, h_p \uplus h) \xrightarrow{0} \text{skip, (s, h_p \uplus h), ok}$, by definition we have $\text{assume}(B), (s, h_p \uplus h) \xrightarrow{1} \text{skip, (s, h_p \uplus h), ok}$. Consequently, we have $(s, h_p) \in p \land B, s \xrightarrow{\text{mod}(\text{assume}(B))} s$ and $\text{assume}(B), (s, h_p \uplus h) \xrightarrow{1} \text{skip, (s, h_p \uplus h), ok}$, as required.

\[\text{Case Assume}\]

This rule can be immediately derived from Assume (proved above) by picking $p \equiv \text{true}$.

\[\text{Case Error}\]

Pick arbitrary $p$ such that $T_B \vdash [p] \text{ error} [err: p]$. Pick an arbitrary $(s, h_p) \in p$ and an arbitrary $h$ such that $h_p \neq h$. Let $\sigma = (s, h_p \uplus h)$. From S-Error we then have error, $\sigma \rightarrow \text{skip, } \sigma, \text{ err}$. As such, since $(s, h_p) \in p$, by definition we have error, $\sigma \xrightarrow{1} \text{skip, } \sigma, \text{ err}$, as required. Moreover, as $\text{mod}(\text{error}) = \emptyset$ we also have $s \xrightarrow{\text{mod}(\text{error})} s$, as required.
**Case** Seq

Pick arbitrary \( p, q, r, C_1, C_2, \varepsilon \) such that \( \vdash_B [p] C_1 \mid [\varepsilon: r] \) and \( \vdash_B [r] C_2 \mid [\varepsilon: q] \). Pick an arbitrary \((s, h_p) \in p\) and an arbitrary \( h \) such that \( h_p \neq h \). From \( \vdash_B [p] C_1 \mid [\varepsilon: r] \) and the inductive hypothesis we then know there exists \((s_r, h_r) \in r, i\) such that \( s \sim_{\text{mod}(C_1)} s_r \) and \( C_1, (s, h_p \uplus h) \vdash^\downarrow -, (s_r, h_r \uplus h), \text{ok} \).

Moreover, as \((s_r, h_r) \in r, i\), from \( \vdash_B [r] C_2 \mid [\varepsilon: q] \) and the inductive hypothesis we know there exists \((s', h_q) \in q, j\) such that \( s \sim_{\text{mod}(C_2)} s' \) and \( C_2, (s, h_r \uplus h) \vdash^\downarrow -, (s', h_q \uplus h), \varepsilon \). As \( s \sim_{\text{mod}(C_1)} s_r \) and \( s_r \sim_{\text{mod}(C_2)} s' \), by definition we also have \( s \sim_{\text{mod}(C_1;C_2)} s_r \) and \( s_r \sim_{\text{mod}(C_1;C_2)} s' \), and thus we also have \( s \sim_{\text{mod}(C_1;C_2)}s' \). On the other hand, as \( C_1, (s, h_p \uplus h) \vdash^\downarrow -, (s_r, h_r \uplus h), \text{ok} \) and \( C_2, (s, h_r \uplus h) \vdash^\downarrow -, (s', h_q \uplus h), \varepsilon \).

That is, there exists \((s', h_q) \in q, n\) such that \( s \sim_{\text{mod}(C_1;C_2)} s' \), \( C_1; C_2, (s, h_p \uplus h) \vdash^\downarrow -, (s', h_q \uplus h), \varepsilon \), as required.

**Case** SeqER

Pick arbitrary \( p, q, C_1, C_2 \) such that \( \vdash_B [p] C_1; C_2 \mid [\varepsilon: q] \). Pick an arbitrary \((s, h_p) \in p\) and an arbitrary \( h \) such that \( h_p \neq h \). From the \( \vdash_B [p] C_1 \mid [\varepsilon: q] \) premise and the inductive hypothesis we then know there exists \((s', h_q) \in q, i\) such that \( s \sim_{\text{mod}(C_1)} s' \) and \( C_1, (s, h_p \uplus h) \vdash^\downarrow -, (s', h_q \uplus h), \varepsilon \).

As such, from Lemma 3 we know \( C_1; C_2, (s, h_p \uplus h) \vdash^\downarrow -, (s', h_q \uplus h), \text{er} \), as required.

**Case** Choice

Pick arbitrary \( p, q, C_1, C_2, \varepsilon \) and \( i \in \{1,2\} \) such that \( \vdash_B [p] C_1 + C_2 \mid [\varepsilon: q] \). Pick an arbitrary \((s, h_p) \in p\) and an arbitrary \( h \) such that \( h_p \neq h \). From the \( \vdash_B [p] C_i \mid [\varepsilon: q] \) premise and the inductive hypothesis we then know there exists \((s', h_q) \in q, i\) such that \( s \sim_{\text{mod}(C_i)} s' \) and \( C_i, (s, h_p \uplus h) \vdash^\downarrow -, (s', h_q \uplus h), \varepsilon \).

Moreover, from S-Choice we have \( C_1 + C_2, (s, h_p \uplus h) \rightarrow C_i, (s, h_p \uplus h), \text{ok} \). As such, from the definition of \( \rightarrow^i \) we have \( C_1 + C_2, (s, h_p \uplus h) \rightarrow^i \rightarrow^\downarrow -, (s', h_q \uplus h), \varepsilon \), as required.

**Case** Loop0

Pick arbitrary \( p, C \) such that \( \vdash_B [p] C^* \mid [\varepsilon: p] \). Pick an arbitrary \((s, h_p) \in p\) and an arbitrary \( h \) such that \( h_p \neq h \). From S-Loop0 we have \( C^*, (s, h_p \uplus h) \rightarrow \text{skip}, (s, h_p \uplus h), \text{ok} \). As such, as we have skip, \( (s, h_p \uplus h) \rightarrow_0 \text{skip}, (s, h_p \uplus h), \text{ok} \) (from the definition of \( \rightarrow_0 \)), by definition we have \( C^*, (s, h_p \uplus h) \rightarrow^\downarrow \text{skip}, (s, h_p \uplus h), \text{ok} \). Moreover, by definition we have \( s \sim_{\text{mod}(C^*)} s \), as required.

**Case** Loop

Pick arbitrary \( p, C, q \) such that \( \vdash_B [p] C^* \mid [\varepsilon: q] \). Pick an arbitrary \((s, h_p) \in p\) and an arbitrary \( h \) such that \( h_p \neq h \). From the \( \vdash_B [p] C^*; C \mid [\varepsilon: q] \) premise and the inductive hypothesis we know there exists \((s', h_q) \in q, j\) such that \( s \sim_{\text{mod}(C^*;C)} s' \) and \( C^*; C, (s, h_p \uplus h) \vdash^\downarrow -, (s', h_q \uplus h), \varepsilon \).

As such, from the definition of \( \rightarrow^i \) we have \( C^*, (s, h_p \uplus h) \rightarrow^i \rightarrow^\downarrow -, (s', h_q \uplus h), \varepsilon \), as required.
Case Loop-Subvariant
Pick $p, C, k$ such that $\vdash_B \left[ p(0) \right] C^* \left[ ok : p(k) \right]$. Pick arbitrary $(s, h_p) \in p(0)$ and $h$ such that $h_p \neq h$.

From the $\forall n \in \mathbb{N} \vdash_B \left[ p(n) \right] C \left[ ok : p(n+1) \right]$ premise and the inductive hypothesis we know:

$$\forall n \in \mathbb{N}, (s, h_p) \in p(n), h, h_p \neq h \Rightarrow \exists (s', h_q) \in p(n+1), j. s \sim_{\mod(C^*)} s' \land C, (s, h_p \equiv h) \rightarrow \neg, (s', h_q \equiv h), ok$$

Consequently, from Lemma 16 we know there exists $(s', h_q) \in p(k)$ and $j$ such that $s \sim_{\mod(C^*)} s'$ and $C^*, (s, h_p \equiv h) \rightarrow \neg, (s', h_q \equiv h), ok$, as required.

Case Local
Pick arbitrary $p, C, q, e$ such that $\vdash_B \left[ \exists x. p \right]$ local $x$ in $C \left[ e : \exists x. q \right]$. Pick an arbitrary $(s, h_p) \in \exists x. p$ and an arbitrary $h$ such that $h_p \neq h$; i.e. there exists $v, s_p$ such that $s_p = s[x \mapsto v]$ and $(s_p, h_p) \in p$.

From the $\vdash_B \left[ p \right] C \left[ e : q \right]$ premise and the inductive hypothesis we know there exists $(q, h_q) \in q$ and $n$ such that $s_p \sim_{\mod(C)} s_q$ and $C, (s_p, h_p \equiv h) \rightarrow^n, (q, h_q \equiv h), e$. From S-Local we have local $x$ in $C$, $(s, h_p \equiv h) \rightarrow C; \text{end}(x, s(x))$, $(s_p, h_p \equiv h)$. There are now two cases to consider: 1) $e = \text{ok}$; or 2) $e = \text{err}$.

In case (1), let $s'' = s_q[x \mapsto s(x)]$. Consequently, as $s_p \sim_{\mod(C)} s_q$ and $s''(x) = s(x)$, from the definitions of $s_p$ and $s''$ we also have $s \sim_{\text{local } x \in C} s''$. From S-LocalEnd we then have

$$\text{end}(x, s(x)), (s_q, h_q \equiv h) \rightarrow \text{skip}, (s'', h_q \equiv h).$$

From the definition of $\rightarrow^0$ we have skip, $(s'', h_q \equiv h)$, $\rightarrow^0$ skip, $(s'', h_q \equiv h)$, $\rightarrow^1$ skip, $(s'', h_q \equiv h)$. Consequently, since we also have $C, (s_p, h_p \equiv h) \rightarrow^n, (q, h_q \equiv h), e$, from Lemma 2 we know there exists $m$ such that $C; \text{end}(x, s(x)), (s_p, h_p \equiv h) \rightarrow^m \text{skip}, (s'', h_q \equiv h), \text{ok}$. On the other hand, since we have local $x$ in $C$, $(s, h_p \equiv h) \rightarrow C; \text{end}(x, s(x)), (s_p, h_p \equiv h), \text{by definition of } \rightarrow^{m+1}$ we also have local $x$ in $C, (s, h_p \equiv h) \rightarrow^{m+1} \text{skip}, (s'', h_q \equiv h), \text{ok}$. Finally, as $(q, h_q) \in q$ and $s'' = s_q[x \mapsto s(x)]$, by definition we also have $(s'', h_q) \in \exists x. q$, as required.

In case (2), from $C, (s_p, h_p \equiv h) \rightarrow^n, (q, h_q \equiv h), e$ and Lemma 3 we have $C; \text{end}(x, s(x)), (s_p, h_p \equiv h) \rightarrow^n \neg, (q, h_q \equiv h), e$. On the other hand, since we have local $x$ in $C$, $(s, h_p \equiv h) \rightarrow C; \text{end}(x, s(x)), (s_p, h_p \equiv h)$, by definition of $\rightarrow^n$ we also have local $x$ in $C, (s, h_p \equiv h) \rightarrow^{n+1} \neg, (q, h_q \equiv h), e$. Moreover, as $s_p = s[x \mapsto q]$, mod(local $x$ in $C$) = mod($C$) $\cup \{x\}$ and $s_p \sim_{\text{mod(local } x \in C)} s_q$, by definition we also have $s \sim_{\text{mod(local } x \in C)} s_q$. Finally, as $(q, h_q) \in q$, by definition we also have $(s_q, h_q) \in \exists x. q$, as required.

Case Disj
Pick arbitrary $p_1, p_2, q_1, q_2, C$ such that $\vdash_B \left[ p_1 \lor p_2 \right] C \left[ e : q_1 \lor q_2 \right]$. Pick an arbitrary $(s, h_p) \in p_1 \lor p_2$ and an arbitrary $h$ such that $h_p \neq h$. There are then two cases to consider: 1) $(s, h_p) \in p_1$; or 2) $(s, h_p) \in p_2$.

In case (1), from the $\vdash_B \left[ p_1 \right] C \left[ e : q_1 \right]$ premise and the inductive hypothesis we know there exists $(s', h_q) \in q_1$, $n$ such that $s \sim_{\mod(C)} s', C, (s, h_p \equiv h) \rightarrow^n \neg, (s', h_q \equiv h), e$. That is, there exists $(s', h_q) \in q_1 \lor q_2$ and $n$ such that $s \sim_{\mod(C)} s', C, (s, h_p \equiv h) \rightarrow^n \neg, (s', h_q \equiv h), e$, as required. The proof of case (2) is analogous and omitted.

Case DisjTrack
Pick arbitrary $P_1, P_2, Q_1, Q_2, C$ such that $\vdash_B \left[ P_1 \lor P_2 \right] C \left[ e : Q_1 \lor Q_2 \right]$. Pick an arbitrary $i \in \text{dom}(P_1 \lor$
$P_2$), $(s, h_p) \in (P_1 \times P_2)(i)$ and an arbitrary $h$ such that $h_p \neq h$. We then know that either $i \in \text{dom}(P_1)$ or $i \in \text{dom}(P_2)$. Without loss of generality, let us assume $i \in \text{dom}(P_1)$.

As $(s, h_p) \in (P_1 \times P_2)(i)$ and $i \in \text{dom}(P_1)$, we then have $(s, h_p) \in P_1(i)$. From the $\vdash_B [P_1] C [\epsilon : Q_1]$ premise, the definition of merged triples premise and the inductive hypothesis we know there exists $(s', h_q) \in Q_1(i)$, $n$ such that $s \xrightarrow{\text{mod}(C)} s'$ and $C, (s, h_p \uplus h) \xrightarrow{n} - , (s', h_q \uplus h), \epsilon$. That is, there exists $(s', h_q) \in (Q_1 \cup Q_2)(i)$ and $n$ such that $s \xrightarrow{\text{mod}(C)} s'$ and $C, (s, h_p \uplus h) \xrightarrow{n} - , (s', h_q \uplus h), \epsilon$, as required.

**Case Cons**

Pick arbitrary $P, Q, C, I$ such that $\vdash_B [P \downarrow I] C [\epsilon : Q \downarrow I]$. Pick an arbitrary $i \in \text{dom}(P \downarrow I)$; that is, from the $I \subseteq \text{dom}(P)$ we know $i \in \text{dom}(P \cap I)$, i.e. $i \in \text{dom}(P)$ and $i \in I$. Pick an arbitrary $(s, h_p) \in P(i)$ and an arbitrary $h$ such that $h_p \neq h$. From the $\vdash_B [P] C [\epsilon : Q]$ premise, the definition of merged triples and the inductive hypothesis we know there exists $(s', h_q) \in Q(i)$ and $n$ such that $s \xrightarrow{\text{mod}(C)} s'$ and $C, (s, h_p \uplus h) \xrightarrow{n} - , (s', h_q \uplus h), \epsilon$. As $i \in I$ and $i \in \text{dom}(Q)$, we know $i \in \text{dom}(Q \downarrow I)$. That is, there exists $i \in \text{dom}(Q \downarrow I)$, $(s', h_q) \in (Q \downarrow I)(i)$ and $n$ such that $s \xrightarrow{\text{mod}(C)} s'$ and $C, (s, h_p \uplus h) \xrightarrow{n} - , (s', h_q \uplus h), \epsilon$, as required.

**Case Alloc**

Pick arbitrary $x, v$ and $(s, h_p) \in p$ and $h$ such that $h_p \neq h$. We then know $h_p \neq \emptyset$. Pick $l$ such that $l \notin \text{dom}(h)$ and let $h_q = [l \mapsto v]$ and $s' = s[x \mapsto l]$; as such, we also have $(s', h_q) \in l \mapsto v \ast x = l$ and $s \xrightarrow{\text{mod}(x = \text{alloc}(l))} s'$. Since $l \notin \text{dom}(h)$ and $h_q = [l \mapsto v]$, by definition of # we also know $h_q \neq h$.

From SL-ALLOC we then have $x \leftarrow \text{alloc}(), (s, h_p \uplus h) \rightarrow \text{skip}, (s', h_q \uplus h), \text{ok}$, and since we also have skip, $(s', h_q \uplus h) \xrightarrow{0} \text{skip}, (s', h_q \uplus h), \text{ok}$, by definition of $\rightarrow$ we have $x \leftarrow \text{alloc}(), (s, h_p \uplus h) \xrightarrow{0} \text{skip}, (s', h_q \uplus h), \text{ok}$, as required.

**Case AllocFree**

Pick arbitrary $x, y$ and $(s, h_p) \in p$ and $h$ such that $h_p \neq h$. We then know there exists $l$ such that $s(y) = l$ and $h_p \models [l \mapsto \bot]$. Let $h_q = [l \mapsto v]$ and $s' = s[x \mapsto l]$; as such, we also have $(s', h_q) \in y \leftarrow v \ast x = y$ and $s \xrightarrow{\text{mod}(x = \text{alloc}(l))} s'$. Since $h_p \neq h$ and $\text{dom}(h_p) = \text{dom}(h_q)$, by definition of # we also know $h_q \neq h$. From SL-ALLOC we then have $x \leftarrow \text{alloc}(), (s, h_p \uplus h) \rightarrow \text{skip}, (s', h_q \uplus h), \text{ok}$, and since we also have skip, $(s', h_q \uplus h) \xrightarrow{0} \text{skip}, (s', h_q \uplus h), \text{ok}$, by definition of $\rightarrow$ we have $x \leftarrow \text{alloc}(), (s, h_p \uplus h) \xrightarrow{0} \text{skip}, (s', h_q \uplus h), \text{ok}$, as required.

**Case Free**

Pick an arbitrary $x$ and $(s, h_p) \in p$ and $h$ such that $h_p \neq h$. We then know there exists $l, v$ such that $s(x) = l$ and $h_p \models [l \mapsto v]$. Let $h_q = [l \mapsto \bot]$; we then have $(s, h_q) \in x \not\rightarrow$ and $s \xrightarrow{\text{mod}(\text{free}(x))} s$. Since $h_p \neq h$ and $\text{dom}(h_p) = \text{dom}(h_q)$, from the definition of $\not\rightarrow$ we also know that $h_q \neq h$. From SL-FREE we then have free$(x)$, $(s, h_p \uplus h) \rightarrow \text{skip}, (s, h_q \uplus h), \text{ok}$, and since we also have skip, $(s, h_q \uplus h) \xrightarrow{0} \text{skip}, (s, h_q \uplus h), \text{ok}$, by definition of $\rightarrow$ we have free$(x)$, $(s, h_p \uplus h) \xrightarrow{0} \text{skip}, (s, h_q \uplus h), \text{ok}$, as required.

**Case FreeER**

Pick an arbitrary $x$ and $(s, h_p) \in p$ and $h$ such that $h_p \neq h$. We then know there exists $l$ such that $s(x) = l$ and $h_p \models [l \mapsto \bot]$. Let $h_q = h_p$; we then have $(s, h_q) \in x \not\rightarrow$ and $s \xrightarrow{\text{mod}(\text{free}(x))} s$. From

```plaintext
% implemented
```
we then have free(x), (s, h_p \uplus h) \rightarrow skip, (s, h_q \uplus h), er, as required.

Case FreeNull
Pick an arbitrary x and (s, h_p) \in p and h such that h_p \neq h. We then know s(x) = null and h_p \perp \emptyset. Let h_q = h_p; we then have (s, h_q) \in x = null and s \overset{\text{\text{nmod}}}{}^{} s. From SL-FreeEr
we then have free(x), (s, h_p \uplus h) \rightarrow skip, (s, h_q \uplus h), er, and thus by definition of \rightarrow we have free(x), (s, h_p \uplus h) \overset{1}{} \rightarrow skip, (s, h_q \uplus h), er, as required.

Case Store
Pick an arbitrary x and (s, h_p) \in p and h such that h_p \neq h. We then know there exists l, v, v_y such that s(x) = l, s(y) = v_y, s(e) = v and h_p \perp [l \leftrightarrow v]. Let h_q = [l \leftrightarrow v_y]; we then have (s, h_q) \in x \leftrightarrow y and s \overset{\text{\text{nmod}}}{}^{} s. Since h_p \neq h and dom(h_p) = dom(h_q), from the definition of \uplus we also know that h_q \neq h. From SL-Store we then have [x] := y, (s, h_p \uplus h) \rightarrow skip, (s, h_q \uplus h), ok, and since we also have skip, (s, h_q \uplus h) \overset{0}{} \rightarrow skip, (s, h_q \uplus h), ok, by definition of \rightarrow we have [x] := y, (s, h_p \uplus h) \overset{1}{} \rightarrow skip, (s, h_q \uplus h), ok, as required.

Case StoreEr
Pick an arbitrary x and (s, h_p) \in p and h such that h_p \neq h. We then know there exists l such that s(x) = l and h_p \perp [l \leftrightarrow \bot]. Let h_q = h_p; we then have (s, h_q) \in x \not\leftrightarrow and s \overset{\text{\text{nmod}}}{}^{} s. From SL-StoreEr we then have [x] := y, (s, h_p \uplus h) \rightarrow skip, (s, h_q \uplus h), er and thus by definition of \rightarrow we have [x] := y, (s, h_p \uplus h) \overset{1}{} \rightarrow skip, (s, h_q \uplus h), er, as required.

Case StoreNull
Pick an arbitrary x and (s, h_p) \in p and h such that h_p \neq h. We then know s(x) = null and h_p \perp \emptyset. Let h_q = h_p; we then have (s, h_q) \in x = null and s \overset{\text{\text{nmod}}}{}^{} s. From SL-StoreEr
we then have [x] := y, (s, h_p \uplus h) \rightarrow skip, (s, h_q \uplus h), er, and thus by definition of \rightarrow we have [x] := y, (s, h_p \uplus h) \overset{1}{} \rightarrow skip, (s, h_q \uplus h), er, as required.

Case Load
Pick arbitrary x and (s, h_p) \in p and h such that h_p \neq h. We then know there exists l, v, v_x such that s(x) = s(x') = v_x, s(y) = l, s(e) = v and h_p \perp [l \leftrightarrow v]. Let h_q = h_p and s' = s[x \leftrightarrow v]; as such, we also have (s', h_q) \in x = e[x'/x] * y \leftrightarrow e[x'/x] and s \overset{\text{\text{nmod}}}{}^{} s'. Since h_p \neq h and
h_p = h_q, we also know h_q \neq h. From SL-Load we then have x := [y], (s, h_p \uplus h) \rightarrow skip, (s', h_q \uplus h), ok, and since we also have skip, (s', h_q \uplus h) \overset{0}{} \rightarrow skip, (s', h_q \uplus h), ok, by definition of \rightarrow we have
x := [y], (s, h_p \uplus h) \overset{1}{} \rightarrow skip, (s', h_q \uplus h), ok, as required.

Case Loader
Pick an arbitrary y and (s, h_p) \in p and h such that h_p \neq h. We then know there exists l such that
s(y) = l and h_p \perp [l \leftrightarrow \bot]. Let h_q = h_p; we then have (s, h_q) \in y \not\leftrightarrow and s \overset{\text{\text{nmod}}}{}^{} s. From SL-Loader we then have x := [y], (s, h_p \uplus h) \rightarrow skip, (s, h_q \uplus h), er and thus by definition of \rightarrow we
Case LoadNull

Pick an arbitrary $y$ and $(s, h_p) \in p$ and $h$ such that $h_p \neq h$. We then know $s(y) = \text{null}$ and $h_p \neq \emptyset$. Let $h_q = h_p$; we then have $(s, h_q) \in y = \text{null}$ and $s \vdash_{\text{mod}(x := \lfloor y \rfloor)} s$. From SL-Loader we then have $x := \lfloor y \rfloor, (s, h_p \uplus h) \xrightarrow{1} \text{skip}, (s, h_q \uplus h), \text{er}$, and thus by definition of $\xrightarrow{1}$ we have $x := \lfloor y \rfloor, (s, h_p \uplus h) \xrightarrow{1} \text{skip}, (s, h_q \uplus h), \text{er}$, as required.

Case Frame

Pick arbitrary $(s_1, h_1) \in p \times r$ and $h$ such that $h_1 \neq h$. From the definition of * we then know there exists $h_p, h_r$ such that $(s_1, h_p) \in p$, $(s_1, h_r) \in r$ and $h_1 \neq h_p \uplus h_r$. From the definition of $\uplus$ and $\时候$ we then also have $h_p \neq h_r \uplus h$. On the other hand, from the premise of Frame we have $\vdash_B [p] C [\epsilon : q]$ and thus from the inductive hypothesis we know there exists $s_2, h_q, n$ such that $s_1 \vdash_{\text{mod}(C)} s_2$, $(s_2, h_q) \in C$, $(s_1, h_p \uplus h_r \uplus h) \xrightarrow{n} \neg, (s_2, h_q \uplus h_r \uplus h), \epsilon$. Moreover, since $s_1 \vdash_{\text{mod}(C)} s_2$ and as from the premise of Frame we have $\text{mod}(C) \cap \text{fv}(r) = \emptyset$, we also have $s_1 \vdash_{\text{mod}(C)} s_2$. Consequently, since $(s_1, h_1) \in r$, from Prop. 19 we have $(s_2, h_r) \in r$. As such from the definition of * we have $(s_2, h_q \uplus h_r) \in q \times r$. That is, we know there exists $s_2$ and $h_2 = h_q \uplus h_r$ such that $s_1 \vdash_{\text{mod}(C)} s_2$, $(s_2, h_2) \in q \times r$ and $C, (s_1, h_1 \uplus h) \xrightarrow{n} \neg, (s_2, h_2 \uplus h), \epsilon$, as required. □

Lemma 18 (BUA soundness in UNTER$^\text{SL}$). For all $p, C, q, \epsilon$, if $\vdash_B [p] C [\epsilon : q]$ is derivable using the rules in Fig. 9, then $\models_B [p] C [\epsilon : q]$ holds.

Proof. Pick arbitrary $p, C, q, \epsilon$ such that $\vdash_B [p] C [\epsilon : q]$ is derivable using the rules in Fig. 9. Pick an arbitrary $(s_p, h_p) \in p$. It then suffices to show there exists $(s_q, h_q) \in q$ and $n$ such that $C, (s_p, h_p) \xrightarrow{n} \neg, (s_q, h_q), \epsilon$.

Let $h_0 = \emptyset$ denote the empty heap (with an empty domain). From the definition of $\uplus$ and $\时候$ we then know that $h_p \neq h_0$. As such, from Lemma 17 we know there exists $(s_q, h_q) \in q$ and $n$ such that $C, (s_p, h_p \uplus h_0) \xrightarrow{n} \neg, (s_q, h_q \uplus h_0), \epsilon$. That is, there exists $(s_q, h_q) \in q$ and $n$ such that $C, (s_p, h_p) \xrightarrow{n} \neg, (s_q, h_q), \epsilon$, as required. □

F.2 FUA Soundness in UNTER$^\text{SL}$

Lemma 19. For all $p, C, q, \epsilon$, if $\vdash_B [p] C [\epsilon : q]$ can be derived using the proof rules in Fig. 9, then:

$$\forall (s_q, h_q) \in q. \forall h. h_q \# h \implies \exists (s_p, h_p) \in p. n. s_p \vdash_{\text{mod}(C)} s_q \land C, (s_p, h_p \uplus h) \xrightarrow{n} \neg, (s_q, h_q \uplus h), \epsilon$$

Proof. The proof of this lemma is analogous to that of Lemma 17 and is omitted. □

Lemma 20 (FUA soundness in UNTER$^\text{SL}$). For all $p, C, q, \epsilon$, if $\vdash_B [p] C [\epsilon : q]$ is derivable using the rules in Fig. 9, then $\models_B [p] C [\epsilon : q]$ holds.

Proof. Pick arbitrary $p, C, q, \epsilon$ such that $\vdash_B [p] C [\epsilon : q]$ is derivable using the rules in Fig. 9. Pick an arbitrary $(s_q, h_q) \in q$. It then suffices to show there exists $(s_p, h_p) \in p$ and $n$ such that $C, (s_p, h_p) \xrightarrow{n} \neg, (s_q, h_q), \epsilon$.

Let $h_0 = \emptyset$ denote the empty heap (with an empty domain). From the definition of $\uplus$ and $\时候$ we then know that $h_q \# h_0$. As such, from Lemma 19 we know there exists $(s_p, h_p) \in p$ and $n$ such that $C, (s_p, h_p \uplus h_0) \xrightarrow{n} \neg, (s_q, h_q \uplus h_0), \epsilon$. That is, there exists $(s_p, h_p) \in p$ and $n$ such that $C, (s_p, h_p) \xrightarrow{n} \neg, (s_q, h_q), \epsilon$, as required. □
F.3 Divergent Soundness in \textsc{unter}κ

Lemma 21. For all \( C, \sigma, C', \sigma', \epsilon, n \), if \( n > 0 \) and \( C, \sigma \xrightarrow{n} C', \sigma', \epsilon \), then \( C, \sigma \xrightarrow{n} C', \sigma', \epsilon \).

Proof. The proof of this lemma is analogous to that of Lemma 9 and is omitted. \( \square \)

Lemma 22. For all \( C_1, C_2, C'_1, \sigma, C', \epsilon, C' \), if \( C_1, \sigma \xrightarrow{i} -, \sigma', \epsilon \) and \( C_2, \sigma' \xrightarrow{j} C' \), then \( C_1; C_2, C'_1; C_2, C', \epsilon \).

Proof. The proof of this lemma is analogous to that of Lemma 10 and is omitted. \( \square \)

Lemma 23. For all \( \sigma, \sigma', \sigma'', C, C_1, C_2, C', \epsilon \), if \( C, \sigma \xrightarrow{i} C_1, \sigma', \epsilon \) and \( C_2, \sigma'' \xrightarrow{j} C' \), then there exists \( n \) such that \( C_1; C_2, C'_1; C_2, C', \epsilon \).

Proof. The proof of this lemma is analogous to that of Lemma 11 and is omitted. \( \square \)

Lemma 24. For all \( i, j, C, C', C'', s, s', s'', \epsilon \), if \( C, s \xrightarrow{i} C', s', \epsilon \) and \( C', s'' \xrightarrow{j} C$, s', \epsilon \), then \( C, s \xrightarrow{i+j} C', s' \).

Proof. The proof of this lemma is analogous to that of Lemma 12 and is omitted. \( \square \)

Lemma 25. For all \( p, C \), if \( \vdash [p] C \infty \) can be derived using the proof rules in Fig. 9, then:

\[
\forall \sigma_p \in p. \forall \sigma_p \neq \sigma \implies \exists C_1, \sigma_1, C_2, \sigma_2, \ldots. C, \sigma_p \circ \sigma \xrightarrow{+} C_1, \sigma_1, ok \xrightarrow{+} C_2, \sigma_2, ok \xrightarrow{+} \ldots
\]

Proof. By induction on the structure of the divergence rules in Fig. 3 and Fig. 9.

Case \textsc{div-seq1}.

Pick arbitrary \( p, C_1, C_2 \) such that \( [p] C_1; C_2 \infty \). Pick an arbitrary \( \sigma_p \in p \) and \( \sigma \) such that \( \sigma_p \neq \sigma \).

From the \( [p] C_1 \infty \) premise and the inductive hypothesis we know there exists an infinite series \( C'_1, C'_2, \ldots \) and \( \sigma_1, \sigma_2, \ldots \), such that \( C_1, \sigma_p \circ \sigma \xrightarrow{n} C'_1, \sigma_1, ok \xrightarrow{+} C'_2, \sigma_2, ok \xrightarrow{+} \ldots \). As such, from the definition of \( \xrightarrow{+} \) and Lemma 22 we have \( C_1; C_2, \sigma_p \circ \sigma \xrightarrow{+} C'_1; C_2, \sigma_2, ok \xrightarrow{+} C'_3; C_2, \sigma_3, ok \xrightarrow{+} \ldots \), as required.

Case \textsc{div-seq2}.

Pick arbitrary \( p, q, C_1, C_2 \) such that \( [p] C_1; C_2 \infty \). Pick an arbitrary \( \sigma_p \in p \) and \( \sigma \) such that \( \sigma_p \neq \sigma \). From the \( \vdash_B [p] C_1 \infty \) premise and Lemma 17 we know there exists \( \sigma_q \in q \) and \( n \) such that \( C_1, \sigma_p \circ \sigma \xrightarrow{+} C'_1, \sigma_1, ok \xrightarrow{+} C'_2, \sigma_q \circ \sigma, ok \) and \( C_2, \sigma_q \circ \sigma \xrightarrow{+} C'_3, \sigma_3, ok \xrightarrow{+} \ldots \). As such, from the definition of \( \xrightarrow{+} \) and Lemma 17 we have \( C_1; C_2, \sigma_p \circ \sigma \xrightarrow{+} C'_1; C_2, \sigma_q \circ \sigma \xrightarrow{+} C'_3, \sigma_3, ok \xrightarrow{+} \ldots \), as required.

Case \textsc{div-choice}.

Pick arbitrary \( p, C_1, C_2 \) such that \( [p] C_1 + C_2 \infty \). Pick an arbitrary \( i \in \{1, 2\} \), \( \sigma_p \in p \) and \( \sigma \) such that \( \sigma_p \neq \sigma \). From the \( [p] C_i \infty \) premise and the inductive hypothesis we know there exists an infinite series \( C_3, C_4, \ldots \) and \( \sigma_3, \sigma_4, \ldots \), such that \( C_1, \sigma_p \circ \sigma \xrightarrow{+} C_3, \sigma_3, ok \xrightarrow{+} C_4, \sigma_4, ok \xrightarrow{+} \ldots \). Moreover, from SL-\textsc{choice} we have \( C_1 + C_2, \sigma_p \circ \sigma \xrightarrow{+} C_1, \sigma_p \circ \sigma, ok \). And thus we have \( C_1 + C_2, \sigma_p \circ \sigma \xrightarrow{+} C_1, \sigma_p \circ \sigma, ok \xrightarrow{+} C_3, \sigma_3, ok \xrightarrow{+} C_4, \sigma_4, ok \xrightarrow{+} \ldots \). That is, by definition of \( \xrightarrow{+} \) we have \( C_1 + C_2, \sigma_p \circ \sigma \xrightarrow{+} C_3, \sigma_3, ok \xrightarrow{+} C_4, \sigma_4, ok \xrightarrow{+} \ldots \), as required.
**Case Div-LoopUnfold**

Pick arbitrary $p, C$ such that $[p] C \rightarrow^{\infty}$. Pick an arbitrary $\sigma_p \in p$ and $\sigma$ such that $\sigma_p \neq \sigma$. From the $[p] C; C^* \rightarrow^{\infty}$ premise and the inductive hypothesis we know there exists an infinite series $C_1, C_2, \cdots$ and $\sigma_1, \sigma_2, \cdots$, such that $C; C^*, \sigma_p \circ \sigma \rightarrow^{+} C_1, \sigma_1, ok \rightarrow^{+} C_2, \sigma_2, ok \rightarrow^{+} \cdots$. Moreover, from SL-Loop we have $C^*, \sigma_p \circ \sigma \rightarrow C; C^*, \sigma_p \circ \sigma, ok$. And thus we have $C^*, \sigma_p \circ \sigma \rightarrow C; C^*, \sigma_p \circ \sigma \rightarrow C; C^*, \sigma_p \circ \sigma, ok \rightarrow^{+} C_1, \sigma_1, ok \rightarrow^{+} C_2, \sigma_2, ok \rightarrow^{+} \cdots$. That is, by definition of $\rightarrow^{+}$ we have $C^*, \sigma_p \circ \sigma \rightarrow^{+} C_1, \sigma_1, ok \rightarrow^{+} C_2, \sigma_2, ok \rightarrow^{+} \cdots$, as required.

**Case Div-LoopNest**

This rule can be derived as follows:

\[
\begin{align*}
[p] C &\rightarrow^{\infty} \quad \text{Div-Seq1} \\
[p] C; C^* &\rightarrow^{\infty} \quad \text{Div-LoopUnfold}
\end{align*}
\]

and thus this rule is sound as we established the soundness of Div-Seq1 and Div-LoopUnfold above.

**Case Div-Loop**

Pick arbitrary $p, C, q$ such that $\vdash [p] C^* \rightarrow^{\infty}$. From SL-Loop we then have:

\[
\forall \sigma_p \in p, \sigma. \exists ! \sigma \in \sigma_p \neq \sigma \Rightarrow C^*, \sigma_p \circ \sigma \rightarrow C; C^*, \sigma_p \circ \sigma, ok
\]  

(14)

From the $\vdash [p] C [ok: q]$ premise, Lemma 17, and the $q \subseteq p$ premise we know $\forall \sigma_p \in p, \sigma. \exists ! \sigma_p' \in p, n. \ C, \sigma_p \circ \sigma \rightarrow^{n} \sigma_p' \circ \sigma, ok$ and thus from Lemma 15 $C, \sigma_p \circ \sigma \rightarrow^{\infty} \sigma_p' \circ \sigma, ok$. That is, from the axiom of choice we know there exist $f : p \rightarrow p$ and $g : p \rightarrow \mathbb{N}$ such that:

\[
\forall \sigma_p \in p, \sigma. \exists ! \sigma \in \sigma_p \neq \sigma \Rightarrow C, \sigma_p \circ \sigma \circ \sigma \circ \sigma \rightarrow^{g(\sigma_p)} \text{skip}, f(\sigma_p) \circ \sigma, ok \land f(\sigma_p) \in p
\]  

(15)

In what follows, we show that $\forall \sigma_p \in p, \sigma. \exists ! \sigma \in \sigma_p \neq \sigma \Rightarrow C^*, \sigma_p \circ \sigma \rightarrow^{\infty} \sigma_p \circ \sigma, ok$.

Pick an arbitrary $\sigma_p \in p$ and $\sigma$ such that $\sigma_p \neq \sigma$. From (2) we have $C, \sigma_p \circ \sigma \rightarrow^{\infty} \text{skip}, f(\sigma_p) \circ \sigma, ok$. There are now two cases to consider: i) $g(\sigma_p) = 0$ or ii) $g(\sigma_p) > 0$. In case (i), we then have $C = \text{skip}$ and $\sigma_p = f(\sigma_p)$. As such, from SL-SeqSkip we have $C; C^*, \sigma_p \circ \sigma \rightarrow C^*, f(\sigma_p) \circ \sigma, ok$, and thus by definition of $\rightarrow^{\infty}$ we have $C; C^*, \sigma_p \circ \sigma \rightarrow^{\infty} C^*, f(\sigma_p) \circ \sigma, ok$.

In case (ii), from $C, \sigma_p \circ \sigma \rightarrow^{\infty} \text{skip}, f(\sigma_p) \circ \sigma, ok$ and Lemma 21 we have $C, \sigma_p \circ \sigma \rightarrow^{\infty}(g(\sigma_p))$ skip, $f(\sigma_p) \circ \sigma, ok$. Consequently, from Lemma 22 we have $C; C^*, \sigma_p \circ \sigma \rightarrow^{\infty}(g(\sigma_p))$ skip, $C^*, f(\sigma_p) \circ \sigma, ok$.

On the other hand, from SL-SeqSkip we have skip; $C^*, f(\sigma_p) \circ \sigma \rightarrow C^*, f(\sigma_p) \circ \sigma, ok$, and thus by definition of $\rightarrow^{\infty}$ we have skip; $C^*, f(\sigma_p) \circ \sigma \rightarrow^{\infty} C^*, f(\sigma_p) \circ \sigma, ok$. From Lemma 24, $C; C^*, \sigma_p \circ \sigma \rightarrow^{\infty}(g(\sigma_p))$ skip; $C^*, f(\sigma_p) \circ \sigma, ok$ and skip; $C^*, f(\sigma_p) \circ \sigma \rightarrow^{\infty} C^*, f(\sigma_p) \circ \sigma, ok$ we know there exists $i$ such that $C; C^*, \sigma_p \circ \sigma \rightarrow^{i} C^*, f(\sigma_p) \circ \sigma, ok$.

That is, in both cases we know there exists $i$ such that $C; C^*, \sigma_p \circ \sigma \rightarrow^{i} C^*, f(\sigma_p) \circ \sigma, ok$.

As such, from (14) and the definition of $\rightarrow^{i+1}$ we have $C^*, \sigma_p \circ \sigma \rightarrow^{i+1} C^*, f(\sigma_p) \circ \sigma, ok$, i.e. $C^*, \sigma_p \circ \sigma \rightarrow^{\infty} C^*, f(\sigma_p) \circ \sigma, ok$. That is, from (15) we have:

\[
\forall \sigma_p \in p, \sigma. \exists ! \sigma \in \sigma_p \neq \sigma \Rightarrow C^*, \sigma_p \circ \sigma \rightarrow^{\infty} C^*, f(\sigma_p) \circ \sigma, ok \land f(\sigma_p) \in p
\]  

(16)

Pick an arbitrary $\sigma_p \in p$ and $\sigma$ such that $\sigma_p \neq \sigma$. From (16) we then know $C^*, \sigma_p \circ \sigma \rightarrow^{\infty} C^*, f(\sigma_p) \circ \sigma, ok \rightarrow^{\infty} \cdots$, as required.
Case $\text{Div-Subvariant}$
Pick arbitrary $p, C, q$ such that $\vdash [p(0)] C^* [\infty]$. From $\text{SL-Loop}$ we then have:

$$\forall \sigma_p \in p, \sigma. \sigma_p \neq \sigma \Rightarrow C^*, \sigma_p \circ \sigma \Rightarrow ; C^*, \sigma_p \circ \sigma \Rightarrow \text{o.k} \quad (17)$$

From the $\forall n \in \mathbb{N}. \vdash [p(n)] C [\text{ok} \vdash p(n+1)]$ premise, Lemma 17, and the $q \leq p$ premise we know

$$\forall n \in \mathbb{N}, \sigma_p \in p(n), \sigma. \sigma_p \neq \sigma \Rightarrow \exists \sigma'_p \in p(n+1), k. C, \sigma_p \circ \sigma \Rightarrow \neg \sigma'_p \circ \sigma, \text{o.k}. $$

That is, from the axiom of choice we know there exists a series of functions, $f_i, g_1, f_2, g_2, \ldots$ such that for each $i \in \mathbb{N}$, we have $f_i : p(i-1) \rightarrow p(i)$ and $g_i : p(i-1) \rightarrow \mathbb{N}$ such that:

$$\forall i \in \mathbb{N}^+. \forall \sigma_p \in p(i-1), \sigma. \sigma_p \neq \sigma \Rightarrow C, \sigma_p \circ \sigma \Rightarrow \text{ok} \wedge f_i(\sigma_p) \in p(i) \quad (18)$$

In what follows, we show that $\forall i \in \mathbb{N}^+. \forall \sigma_p \in p(i-1), \sigma. \sigma_p \neq \sigma \Rightarrow C^*, \sigma_p \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}.$

Pick an arbitrary $i \in \mathbb{N}^+, \sigma_p \in p(i-1)$ and $\sigma$ such that $\sigma_p \neq \sigma$. From (18) we have $C, \sigma_p \circ \sigma \Rightarrow \text{ok}$, skip, $f_i(\sigma_p) \circ \sigma$ and Lemma 21 we have $C, \sigma_p \circ \sigma \Rightarrow \text{ok}$. Consequently, from Lemma 22 we have $C^*, \sigma_p \circ \sigma \Rightarrow g_1(\sigma_p)$, skip, $C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$. On the other hand, from $\text{SL-SeqSkip}$ we have skip, $C^*, f_i(\sigma_p) \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$, and thus by definition of $\sim^1$ we have $C^*, \sigma_p \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$. From Lemma 24, $C^*, \sigma_p \circ \sigma \Rightarrow \text{ok}$, skip, $C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$ and skip, $C^*, f_i(\sigma_p) \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$ we know there exists $j$ such that $C^*, \sigma_p \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$. That is, in both cases we know there exists $j$ such that $C^*, \sigma_p \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$. As such, from (17) and the definition of $\sim^{j+1}$ we have $C^*, \sigma_p \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$, i.e.

$$C^*, \sigma_p \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k} \quad (19)$$

Pick an arbitrary $\sigma_p \in p(0)$ and $\sigma$ such that $\sigma_p \neq \sigma$. From (19) we then know $C^*, \sigma_p \circ \sigma \Rightarrow C^*, f_i(\sigma_p) \circ \sigma, \text{o.k}$, as required.

Case $\text{Div-Cons}$

Pick arbitrary $p, C$ such that $[p] C [\infty]$. Pick an arbitrary $\sigma_p \in p$ and $\sigma$ such that $\sigma_p \neq \sigma$. From the $p \leq p'$ premise we know $\sigma_p \in p'$. As such, from the $[p'] C [\infty]$ premise we know there exists an infinite series $C_1, C_2, \ldots$ and $\sigma_1, \sigma_2, \ldots$, such that $C, \sigma_p \circ \sigma \Rightarrow C_1, \sigma_1, \text{o.k}$, as required.

Case $\text{Div-Frame}$

Pick arbitrary $p, r, C$ such that $[p \ast r] C [\infty]$. Pick an arbitrary $\sigma_{pr} \in p \ast r$ and $\sigma$ such that $\sigma_{pr} \neq \sigma$. As $\sigma_{pr} \in p \ast r$, we know there exist $\sigma_p \in p$ and $\sigma_r \in r$ such that $\sigma_{pr} = \sigma_p \circ \sigma_r$. From the definitions of $\circ$ and $\sigma_{pr}$ and since $\sigma_{pr} \neq \sigma$ we know $\sigma_r \neq \sigma$ and $\sigma_r \neq \sigma$. On the other hand, from the premise of $\text{Div-Frame}$ we have $[p] C [\infty]$ and thus from the inductive hypothesis we know there exists an infinite series $C_1, C_2, \ldots$ and $\sigma_1, \sigma_2, \ldots$, such that $C, \sigma_p \circ \sigma \Rightarrow C_1, \sigma_1, \text{o.k}$, i.e. $C, \sigma_p \circ \sigma \Rightarrow C_1, \sigma_1, \text{o.k}$, as required. □
Proof. Pick arbitrary $p, C$ such that $[p] C [\infty]$ is derivable using the rules in Fig. 3 and Fig. 9. Pick an arbitrary $\sigma_p = (s_p, h_p) \in p$. It then suffices to show there exists an infinite series $C_1, C_2, \ldots$ and $\sigma_1, \sigma_2, \ldots$, such that $C, \sigma_p \rightsquigarrow C_1, \sigma_1, ok \rightsquigarrow C_2, \sigma_2, ok \rightsquigarrow \ldots$.

Let $\sigma_0 = (s_p, h_0)$, where $h_0$ denotes the empty heap (with an empty domain). From the definition of $\circ$ and $\#$ we then know that $\sigma_p \# \sigma_0$. As such, from Lemma 25 we know there exists an infinite series $C_1, C_2, \ldots$ and $\sigma_1, \sigma_2, \ldots$, such that $C, \sigma_p \circ \sigma_0 \rightsquigarrow C_1, \sigma_1, ok \rightsquigarrow C_2, \sigma_2, ok \rightsquigarrow \ldots$. That is, as $\sigma_p \circ \sigma_0 = \sigma_p$, we know there exists an infinite series $C_1, C_2, \ldots$ and $\sigma_1, \sigma_2, \ldots$, such that $C, \sigma_p \rightsquigarrow C_1, \sigma_1, ok \rightsquigarrow C_2, \sigma_2, ok \rightsquigarrow \ldots$, as required. $\square$

Theorem 20 (UNTER\textsuperscript{st} soundness). For all $p, q, C$ and $\epsilon$:

1) if $\vdash_B [p] C [\epsilon : q]$ is derivable using the rules in Fig. 9, then $\models_B [p] C [\epsilon : q]$ holds;

2) if $\vdash_F [p] C [\epsilon : q]$ is derivable using the rules in Fig. 9, then $\models_F [p] C [\epsilon : q]$ holds; and

3) if $\vdash [p] C [\infty]$ is derivable using the rules in Fig. 9, then $\models [p] C [\infty]$ holds.

Proof. The proof of part (1) follows immediately from Lemma 18. The proof of part (2) follows immediately from Lemma 20. The proof of part (3) follows immediately from Lemma 26. $\square$
G NON-TERMINATION CVES

G.1 Network software: Wireshark (C, CVE-2022-3190)

Table 1. Wireshark F5 Ethernet trailer vulnerability (CVE-2022-3190, August 2022). Fix available at https://gitlab.com/wireshark/wireshark/-/merge_requests/7981/diffs. Failure to show parsing progress leads to parsing loop stuck reading the same broken trailer over and over.

```c
static gint
dissect_old_trailer(tvbuff_t *tvb, packet_info *pinfo,
    proto_tree *tree, void *data)
{
    proto_tree *ttree = NULL;
    proto_item *ti = NULL;
    guint off = 0;
    guint read = 0;
    f5eth_tap_data_t *tdata = (f5eth_tap_data_t *)data;
    guint8 type, len, ver;
    while (tvb_reported_length_remaining(tvb, off)) {
        type = tvb_get_guint8(tvb, offset);
        len = tvb_get_guint8(tvb, off + F5_OFF_LENGTH) + F5_OFF_VERSION;
        ver = tvb_get_guint8(tvb, off + F5_OFF_VERSION);
        if (len <= tvb_reported_length_remaining(tvb, offset) &&
            type >= F5TYPE_LOW && type <= F5TYPE_HIGH &&
            len >= F5_MIN_SANE && len <= F5_MAX_SANE &&
            ver <= F5TRAILER_VER_MAX) {
            /* Parse out the specified trailer. */
            switch (type) {
            case F5TYPE_LOW:
                ti = proto_tree_add_item(tree, hf_low_id, tvb,
                    off, len, ENC_NA);
                ttree = proto_item_add_subtree(ti);
                read = dissect_low(tvb, pinfo, ttree,
                    off, len, ver, tdata);
                tdata->trailer_len += read;
                // Bug: next 3 lines should execute after switch
                if (read == 0) {
                    proto_item_set_len(ti, 1);
                    return off;
                }
                break;
            case F5TYPE_MED:
                ti = proto_tree_add_item(tree, hf_med_id, tvb,
                    off, len, ENC_NA);
                ttree = proto_item_add_subtree(ti);
                read = dissect_med(tvb, pinfo, ttree,
                    off, len, ver, tdata);
                tdata->trailer_len += read;
                // Bug: next 3 lines should execute after switch
                if (read == 0) {
                    proto_item_set_len(ti, 1);
                    return off;
                }
                break;
            }
        }
    }
}
```
G.2 Web software: log4j (Java, CVE 2021-45105)

Table 2. A String substitution function is called recursively with a string reference pointing to the string being replaced, leading to an infinite loop. (Java code, CVE 2021-45105). Root cause analysis available at https://www.zerodayinitiative.com/blog/2021/12/17/cve-2021-45105-denial-of-service-via-uncontrolled-recursion-in-log4j-strsubstitutor

```java
// Recursive function that may not terminate
private int substitute(final LogEvent event, final StringBuilder buf, final int offset, final int length, List<String> priorVariables) {
    if (priorVariables == null) {
        priorVariables = new ArrayList<>();
        priorVariables.add(new String(chars, offset, length + lengthChange));
    }
    // Handle cyclic substitution
    if (!priorVariables.contains(varName)) {
        return;
    }
    priorVariables.add(varName);
    String varValue = resolveVariable(event, varName, buf, startPos, endPos);
    // Recursive replace
    final int varLen = varValue.length();
    buf.replace(startPos, endPos, varValue);
    int change = substitute(event, buf, startPos, varLen, priorVariables);
    change = change + (varLen - (endPos - startPos));
    pos += change;
    bufEnd += change;
    lengthChange += change;
    chars = getChars(buf); // in case buffer was altered
    String varNameExpr = new String(chars, startPos + startMatchLen, pos - startPos - startMatchLen);
    // Substitute in variable
    final StringBuilder bufName = new StringBuilder(varNameExpr);
    // Bug: Missing priorVariables param leads to infinite execution
    substitute(event, bufName, 0, bufName.length());
    (...)
```
G.3 Data mining Software: GraphQL (Golang, Sept 2022)

Table 3. Infinite recursion bug in Data Query Language interpreter GraphQL. A parsing lookup table containing function pointers is populated with handlers that can be called recursively while parsing the graph data structure. Bug was fixed in September 2022 to avoid node type confusion when node value string representation is equal to a node type string representation (e.g. String String = "String"). Fix available at https://github.com/solidwall/graphql-go/blob/master/language/parser/parser.go#L843

```go
func init() {
    tokenDefinitionFn = make(map[string]parseDefinitionFn)
    
    tokenDefinitionFn[lexer.BRACE_L.String()] = parseOperationDef
    tokenDefinitionFn[lexer.STRING.String()] = parseTypeSystemDef
    tokenDefinitionFn[lexer.BLOCK_STRING.String()] = parseTypeSystemDef
    tokenDefinitionFn[lexer.NAME.String()] = parseTypeSystemDef
    switch kind := parser.Token.Kind; kind {
    case lexer.BRACE_L, lexer.NAME, lexer.STRING, lexer.BLOCK_STRING:
        item = tokenDefinitionFn[kind.String()]
    case lexer.BRACE_L:
        item = parseOperationDefinition
    case lexer.NAME, lexer.STRING, lexer.BLOCK_STRING:
        item = parseTypeSystemDefinition
    default:
        return nil, unexpected(parser, lexer.Token{})
    }

    if node, err = item(parser); err != nil {
        return nil, err
    }
}

func parseTypeSystemDef(parser *Parser) (ast.Node, error) {
    keywordToken := parser.Token
    var ok bool
    if item, ok = tokenDefinitionFn[keywordToken.Value]; !ok {
        return nil, unexpected(parser, keywordToken)
    }

    // Bug: infinite recursion on parseTypeSystemDef
    return item(parser)
}
```
G.4 System Software: Linux Kernel (C, CVE-2020-25641)


```
+static inline void bvec_iter_skip_zero_bvec(struct bvec_iter *iter)
+{
+    iter->bi_bvec_done = 0;
+    iter->bi_idx++;
+}
+
+#define for_each_bvec(bvl, bio_vec, iter, start)
+    for (iter = (start); (iter).bi_size &&
+        ((bvl = bvec_iter_bvec((bio_vec), (iter))), 1);
+        bvec_iter_advance((bio_vec), &(iter), (bvl).bv_len))
+        (bvl).bv_len ? bvec_iter_advance((bio_vec), &iter),
+            (bvl).bv_len) : bvec_iter_skip_zero_bvec(&iter))
```
G.5  Graphical Software: Blender (C language)

Table 5. Termination bug in graphical software (Blender v3.2). Function blendthumb_extract_from_file_impl contains an infinite loop due to a user-supplied negative stream offset. Fix available at https://developer.blender.org/rB24a2b5cb1292f769dd86e314471443976d5e9512. Table shows minimized vulnerable code.

```c
EThumbStatus blendthumb_extract_from_file_impl(FileReader *file,
                 Thumbnail *thumb,
                 const size_t bhead_size,
                 const bool endian)
{
    uint8_t *bhead_data = BLI_array_alloca(bhead_data, bhead_size);
    while (file_read(file, bhead_data, bhead_size)) {
        int32_t block_size = bytes_to_native_i32(&bhead_data[4], endian);
        switch (*bhead_data) {
            case V: {
                if (!file_seek(file, block_size))
                    return BT_INVALID_THUMB;
            }
        }
    }
}
```
G.6 Machine Learning Software: Sklearn (Python)

Table 6. Termination bug in Machine Learning software (python sklearn version of November 2021). A failing try block prevents the induction variable from being incremented properly. Break in catch block only breaks the inner loop and not the outer one. Fix available at https://github.com/scikit-learn/scikit-learn/pull/21271/commits/325d32fedb48b42faa32b0873a99e9e9f35a125. Table shows minimized vulnerable code.

```python
def discretize(vectors, max_svd_restarts=30, n_iter_max=20):
    svd_restarts = 0
    has_converged = False
    n_samples, n_components = vectors.shape
    while (svd_restarts < max_svd_restarts) and not has_converged:
        n_iter = 0
        while not has_converged:
            n_iter += 1
            vectors_discrete = csc_matrix(np.arange(0, n_samples))
            t_svd = vectors_discrete.T * vectors
            try:
                U, S, Vh = np.linalg.svd(t_svd)
                svd_restarts += 1
            except LinAlgError:
                print("SVD did not converge, try again.")
                break
            if (n_iter > n_iter_max):
                has_converged = True
```
Table 7. Fix for termination bug in OpenSSL. Function BN_mod_sqrt has a non termination condition when computing modular square root arithmetic on a non-prime moduli with invalid curve parameters. Advisory available at https://www.openssl.org/news/secadv/20220315.txt (March 2022).

```c
- /* find smallest i such that b^(2^i) = 1 */
- i = 1;
- if (!BN_mod_sqr(t, b, p, ctx))
-   goto end;
- while (!BN_is_one(t)) {
-   i++;
-   if (i == e) {
-     BNerr(BN_F_BN_MOD_SQRT, BN_R_NOT_A_SQUARE);
-     goto end;
-   }
-   /* Find the smallest i, 0 < i < e, such that b^(2^i) = 1. */
-   for (i = 1; i < e; i++) {
-     if (i == 1) {
-       if (!BN_mod_sqr(t, b, p, ctx))
-         goto end;
-     } else {
-       if (!BN_mod_mul(t, t, t, p, ctx))
-         goto end;
-     }
-   }
-   if (!BN_mod_mul(t, t, t, p, ctx))
-     goto end;
-   if (BN_is_one(t))
-     break;
- }
- /* If not found, a is not a square or p is not prime. */
+ if (i >= e) {
+   BNerr(BN_F_BN_MOD_SQRT, BN_R_NOT_A_SQUARE);
+   goto end;
+ }
```