Non-termination Proving at Scale

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

Program termination is a classic non-safety property whose falsification cannot in general be witnessed by a finite trace. This makes testing for non-termination challenging, and also a natural target for symbolic proof. Several works in the literature apply non-termination proving to small, self-contained benchmarks, but it has not been developed for large, real-world projects; as such, despite its allure, non-termination proving has had limited practical impact. We develop a *compositional* theory for non-termination proving, paving the way for its *scalable* application to large codebases. Discovering non-termination is an under-approximate problem, and we present UNTer, a *sound and complete* under-approximate logic for proving non-termination. We then extend UNTer with separation logic and develop UNTer$^{\text{SL}}$ for heap-manipulating programs, yielding a compositional proof method amenable to automation via under-approximation and bi-abduction. We extend the Pulse analyser from Meta and develop Pulse$^{\infty}$, an automated, compositional prover for non-termination based on UNTer$^{\text{SL}}$. We have run Pulse$^{\infty}$ on large codebases and libraries, each comprising hundreds of thousands of lines of code, including OpenSSL, libxml2, libxpm and CryptoPP; we discovered several previously-unknown non-termination bugs and have reported them to developers of these libraries.

CCS Concepts: • **Theory of computation** → **Programming logic**; **Separation logic**; **Program analysis**; *Program verification*; *Hoare logic*; • **Software and its engineering** → General programming languages.

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

ACM Reference Format:
Azalea Raad, Julien Vanegue, and Peter O'Hearn. 2024. Non-termination Proving at Scale. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 280 (October 2024), 72 pages. https://doi.org/10.1145/3689720

1 Introduction

Why Prove Non-termination? Non-termination (divergence) 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 manually evaluated the bugs in the Common Vulnerabilities and Exposures (CVE) database for security bugs that are due to non-termination, e.g. denial-of-service attacks. We found 916 such CVE's between 2000 and 2022 – see §A. (For ongoing computations such as operating systems, potential non-termination is desirable and unavoidable. Here, we are concerned with buggy, unintended non-termination.)

Authors' Contact Information: Azalea Raad, azalea.raad@imperial.ac.uk, Imperial College London, UK and Bloomberg, USA; Julien Vanegue, jvanegue@bloomberg.net, Bloomberg, USA and Imperial College London, UK; Peter O'Hearn, p.ohearn@ucl.ac.uk, University College London, UK.

ACM 2475-1421/2024/10-ART280
https://doi.org/10.1145/3689720

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 [Calcagno et al. 2011]. Compositionality 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 [Distefano et al. 2019] 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 [Distefano et al. 2019] and a related article from Google [Sadowski et al. 2018] 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) [O'Hearn 2019], 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. [2021] and the 'total Hoare triple' by de Vries and Koutavas [2011].

The *backwards under-approximate* (BUA) triple $\vdash_{\mathrm{B}} [p]\ \mathsf{C}\ [ok\colon q]$ 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): p under-approximates the weakest *possible* precondition, wpp, of C on q: $p \subseteq \mathrm{wpp}(\mathsf{C}, q)$. Here, wpp is the inverse image of the C (relational) semantics, obtained by running Dijkstra's strongest post-condition on the reversal of C.

We next extend our BUA triples with under-approximate *divergence* triples. Specifically, we develop *under-approximate non-termination logic* (UNTER), where we write $\vdash [p]\ \mathsf{C}\ [\infty]$ 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, that each pre-state leads to *some* divergent execution.

We can then state a proof rule for divergence as shown across. The idea behind this rule is simple. As $p \wedge B$ holds initially, after one loop iteration we can get to a state where $p \wedge B$ continues to hold because of the BUA triple in the premise. And in that case we can take one more step, *ad infinitum*.

$$\frac{\vdash_{\mathrm{B}} [p \wedge B]\ \mathsf{C}\ [ok\colon p \wedge B]}{[p \wedge B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\infty]}$$

This proof method is related 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 without returning to the same state twice. But the proof method uses a logical assertion and not a concrete state, and is indeed complete for proving non-termination as we show 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' by Gupta et al. [2008] (see §8 for the relation to their and other work).

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 we 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 §2, §3). 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 $\vdash_{\mathrm{B}} [p]\ \mathsf{C}\ [ok\colon ??]$.

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* [Ishtiaq and O'Hearn 2001], and which is obtained by small changes and a fundamental addition to the existing Pulse program analyser [Le et al. 2022] 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 UNTer$^{\mathrm{SL}}$ as an extension of UNTer (with divergent triples) with separation logic

Our Pulse$^{\infty}$ Prototype. To demonstrate the feasibility of UNTer$^{\mathrm{SL}}$, we have developed Pulse$^{\infty}$, a prototype compositional non-termination prover underpinned by UNTer$^{\mathrm{SL}}$, as an extension of the existing Pulse program analyser (which is underpinned by the ISL theory [Raad et al. 2020] and is compatible with BUA reasoning). To evaluate Pulse$^{\infty}$, we have compared its performance against cutting edge tools such as DynamiTe [Le et al. 2020] by running it on the state-of -the-art non-linear arithmetic extension of the SV-COMP benchmark. While Pulse$^{\infty}$ is not comparable to these tools in the divergence bugs it found, in that Pulse$^{\infty}$ successfully found divergence bugs that were undetected by these tools, while missed others found by these tools (see Table 2 on p. 22), it reported *zero false positives* thanks to its under-approximate nature, in contrast to DynamiTe which suffered several false positives.

More significantly, we have successfully run Pulse$^{\infty}$ on large codebases and libraries, each comprising hundreds of thousands of lines of code (LOC), including OpenSSL, libxml2, CryptoPP and libxpm. To our knowledge, Pulse$^{\infty}$ is *the first automated tool* for detecting divergence bugs in *large code bases and libraries.* As we discuss in the related work (§8), existing tools either focus on a

```
 1 int http_server_get_asn1_req(const ASN1_ITEM *it, ASN1_VALUE **preq,
 2                              char **ppath, BIO **pcbio, BIO *acbio, (...)) {
 3   for (;;) {
 4     char *key, *value;
 5     len = BIO_gets(cbio, inbuf, sizeof(inbuf));
 6     if (len <= 0) goto out;
 7     key = inbuf;
 8     value = strchr(key, ':');
 9     if (value == NULL) goto out;
10     *(value++) = '\0'; }
11 }
```

Listing 1. A divergence bug found by Pulse$^{\infty}$ in OpenSSL

small class of (integer) C programs *without functions calls* (thus excluding libraries), or stipulate the presence of a `main` procedure (once again ruling out libraries), or require that the code under analysis be *confined to a single file* (thus precluding large code bases).

Using Pulse$^{\infty}$, we have automatically analysed each of these libraries *within minutes*. For instance, running Pulse$^{\infty}$ on OpenSSL (804 kLOC) completed *under two minutes* and found four hitherto-unknown divergence bugs. Indeed, we have found new divergence bugs in OpenSSL, libxml2, CryptoPP and libxpm. In the cases of CryptoPP and libxpm we have submitted pull requests with patches. In the case of libxml2, we shared our findings with the development team, who suggested the reported code branches may be unreachable, and thus should be removed entirely from the codebase. For OpenSSL, we present a divergence bug found by Pulse$^{\infty}$ in Listing 1, containing a potential infinite `for` loop on lines 3–10. Specifically, the function shown can keep reading more data with `BIO_gets` (line 5) and never break from the loop. This analysis requires inter-procedural, heap and arithmetic reasoning all at once, which is not uncommon in real-world code. We contacted a senior OpenSSL developer about this bug, who confirmed that the code should be made more restrictive and enforce an upper bound on the amount of data read at this location.

Contributions and Outline. In §2 we present an intuitive overview of BUA and IL reasoning, and describe how we extend them to reason about non-termination. In §3 we present UNTer as a BUA proof system and extend it to account for non-termination, yielding a compositional proof method. In §4 we present several examples of divergence and show how we can detect them using UNTer. In §5 we present the semantic model of UNTer and show that it is *sound* and *complete*. In §6 we develop UNTer^{SL} by extending UNTer with separation logic for heap reasoning. In §7 we extend the under-approximate reasoning framework of Pulse to develop Pulse$^{\infty}$, an automated, compositional prover for non-termination; we evaluate Pulse$^{\infty}$ against other tools in the literature, and report our results of running Pulse$^{\infty}$ on large libraries. We discuss related work in detail in §8.

2 Overview

Incorrectness Logic and Under-Approximate Reasoning. As Godefroid [2005] 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 [O'Hearn 2019; Raad et al. 2020, 2022] and their significant success at finding bugs on an *industrial scale* [Blackshear et al. 2018; Le et al. 2022]. Specifically, Incorrectness logic (IL) [O'Hearn 2019] presented an under-approximate formal foundation for bug detection. It was later extended to enable compositional bug detection in heap-manipulating programs [Raad et al. 2020], and to support concurrency [Raad et al. 2022, 2023]. 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 $\vdash_{\mathsf{F}} [p]$ C $[\epsilon : q]$, where F hints at its *forward* direction, 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 ϵ is 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 (e.g. assert(false)) terminates erroneously and the underlying states are unchanged: $\vdash_{\mathsf{F}} [p]$ 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 $\mathsf{C}_1 + \mathsf{C}_2$ (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 ChoiceF below (left). Similarly, to show a behaviour is possible when executing $\mathsf{C}^\star$ (where $\mathsf{C}^\star$ 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 LoopF below (right), where C^n denotes executing C for n times.

$$\textsc{ChoiceF}\quad \frac{\vdash_{\mathsf{F}} [p]\ \mathsf{C}_i\ [\epsilon : q] \quad \text{for some } i \in \{1, 2\}}{\vdash_{\mathsf{F}} [p]\ \mathsf{C}_1 + \mathsf{C}_2\ [\epsilon : q]} \qquad \textsc{LoopF}\quad \frac{\vdash_{\mathsf{F}} [p]\ \mathsf{C}^n\ [\epsilon : q] \quad \text{for some } n \in \mathbb{N}}{\vdash_{\mathsf{F}} [p]\ \mathsf{C}^\star\ [\epsilon : q]}$$

Non-termination and Under-Approximate Reasoning. Existing literature includes a large body of work [Berdine et al. 2007, 2006; Chawdhary et al. 2008; Cook et al. 2006a,b; da Rocha Pinto et al. 2016; D'Osualdo et al. 2021; Liang and Feng 2016] 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 $S_o \supseteq S$ terminate is sufficient for showing that all traces in S terminate. Showing termination is difficult in the presence of loops: 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. Establishing such invariants and measures is far from straightforward and typically involves reasoning about *ordinal* (rather than natural) numbers.

Showing that a program C does not terminate is compatible with *under-approximate* reasoning: when the traces 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 ***u**nder-approximate, **n**on-**ter**mination 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 $[p]\ \mathsf{C}\ [\infty]$, stating that starting from the states in p program C has divergent (non-terminating) traces. Note that $[p]\ \mathsf{C}\ [\infty]$ 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 $\mathsf{C} \triangleq \mathsf{skip} + (\mathsf{while}\,(\mathsf{true})\ \mathsf{skip})$, the triple $[\mathsf{true}]\ \mathsf{C}\ [\infty]$ 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 $\mathsf{C}_L \triangleq x := 1; \mathsf{while}\,(x > 0)\, x\texttt{++}$ and an arbitrary initial value v, to show that the triple $[x = v]\ \mathsf{C}\ [\infty]$ holds (i.e. C_L does not terminate starting from states satisfying $x = v$), we should ideally show that 1) running $x := 1$ on states in which $x = v$ terminates and modifies the states to those where $x = 1$; and 2) running $\mathsf{while}\,(x > 0)\, x\texttt{++}$ on states where $x = 1$ diverges, i.e. $[x = 1]\ \mathsf{while}\,(x > 0)\, x\texttt{++}\ [\infty]$. 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 $\vdash_{\mathsf{F}} [x = v]\ x := 1\ [ok\colon x = 1]$.

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 $\mathsf{while}\,(B)\ \mathsf{C}$ diverges.

$$\textsc{LoopBad}\ \frac{\vdash_{\mathsf{F}} [p \wedge B]\ \mathsf{C}\ [ok\colon p \wedge B]}{[p \wedge B]\ \mathsf{while}\,(B)\ \mathsf{C}\ [\infty]} \qquad \textsc{LoopFix}\ \frac{\vdash_{\mathsf{B}} [p \wedge B]\ \mathsf{C}\ [ok\colon p \wedge B]}{[p \wedge B]\ \mathsf{while}\,(B)\ \mathsf{C}\ [\infty]}$$

On closer inspection, however, this rule is unsound. Consider the program $\mathsf{while}\,(x > 0)\, x\texttt{--}$; 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 $[x > 0]\ \mathsf{while}\,(x > 0)\, x\texttt{--}\ [\infty]$ is invalid. Nevertheless, we can derive it using LoopBad by showing $\vdash_{\mathsf{F}} [x > 0]\ x\texttt{--}\ [ok\colon x > 0]$. Specifically, the $\vdash_{\mathsf{F}} [x > 0]\ x\texttt{--}\ [ok\colon x > 0]$ 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 $s_q(x) = v$ (i.e. x holds value v in s_q) for some $v > 0$. State s_q is then reachable by running $x\texttt{--}$ on a state $s_p = s_q[x \mapsto v{+}1]$ and $s_p \in x > 0$ (as $v > 0$).

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 $p \wedge B$ to $p \wedge B$ does not yield an infinite execution. Instead, we need a forward chain of C executions from $p \wedge B$ to $p \wedge B$, 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 $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ 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 $p \wedge B$ to some post-state also in $p \wedge B$, then we can repeat this transition infinitely, leading to divergence. Note that in the example above, we cannot show $\vdash_{\mathsf{B}} [x > 0]\ x\texttt{--}\ [ok\colon x > 0]$ (unlike the $\vdash_{\mathsf{F}}$ variant): given state $s_p \in x > 0$ with $s_p(x) = 1$, running $x\texttt{--}$ 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 $[x > 0]\ \mathsf{while}\,(x > 0)\, x\texttt{--}\ [\infty]$. Note that while BUA triples describe *forward reachability*, they denote *backward under-approximation*: $p \subseteq \mathsf{wpp}(\mathsf{C}, q)$, where $\mathsf{wpp}(\mathsf{C}, q)$ 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 $\mathsf{C}_1; \mathsf{C}_2$ does not terminate starting from p, we can show either C_1 does not terminate starting from p (i.e. $[p]\ \mathsf{C}_1\ [\infty]$), or C_1 terminates normally transforming the states to q, and C_2 does not terminate starting from q (i.e. $\vdash_{\mathsf{B}} [p]\ \mathsf{C}_1\ [ok\colon q]$ and $[q]\ \mathsf{C}_2\ [\infty]$). This is captured by the Div-Seq1 and Div-Seq2 rules in Fig. 2 (§3), 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 §3, the BUA rules for reasoning about branches and loops are identical to their FUA counterparts; i.e. the $\vdash_{\mathsf{F}}$ in ChoiceF and LoopF above can simply be replaced with $\vdash_{\mathsf{B}}$ (see Fig. 1). 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 ConsF (for FUA) and ConsB (for BUA) rules in Fig. 1 (p. 10). However, as we describe shortly, in the practical context of industrially-deployed (under-approximate) bug detection tools such as Pulse [Le et al. 2022], 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, ConsF, is in conjunction with the rule of disjunction, Disj in Fig. 1 (p. 10). More concretely, when a given program contains multiple branches, thanks to the 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 Disj. However, when there are many branches (and subsequently many disjuncts in the pre- and post-states), we can simply use ConsF to drop some of the disjuncts in the *post-states*. (Note that using 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 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 $q_1 \vee q_2$ are reached from the pre-states in $p_1 \vee p_2$. As such, when merging the triples $\vdash_{\mathsf{F}} [p_1]\ \mathsf{C}\ [\epsilon : q_1]$ and $\vdash_{\mathsf{F}} [p_2]\ \mathsf{C}\ [\epsilon : q_2]$ into $\vdash_{\mathsf{F}} [p_1 \vee p_2]\ \mathsf{C}\ [\epsilon : q_1 \vee q_2]$, 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*, $P, Q \in \mathbb{N} \xrightarrow{\text{fin}} \mathcal{P}(\textsc{State})$. Intuitively, an indexed disjunction P can be flattened into a standard disjunction as $\bigvee_{i \in dom(P)} P(i)$. We write $[P]\ \mathsf{C}\ [\epsilon : Q]$ as a shorthand for $dom(P) = dom(Q) \wedge \forall i \in dom(P).\ [P(i)]\ \mathsf{C}\ [\epsilon : Q(i)]$, denoting a merged set of triples. Note that a triple $[p]\ \mathsf{C}\ [\epsilon : q]$ can be simply lifted to $[P]\ \mathsf{C}\ [\epsilon : Q]$, where $dom(P) = dom(Q) = \{0\}$ with $P(0) = p$ and $Q(0) = q$. We can then use the DisjTrack rule (Fig. 1 on p. 10) to merge indexed disjuncts – note that the $dom(P_1) \cap dom(P_2) = \emptyset$ premise can be simply satisfied by renaming the domain of P_2. Observe that unlike the Disj rule, DisjTrack is not lossy and preserves the pre-post correspondence. Finally, the unified rule of consequence, Cons (Fig. 1), allows us to drop matching disjuncts from both the pre- and post-states, where $P \downarrow I$ denotes restricting the domain of P to I. The unified Cons rule can be used for both FUA and BUA reasoning.

Unified Triples and Bug Catching Tools. Note that the rules in Fig. 1, excluding ConsB, ConsF and Disj (and instead including Cons and 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 ConsF and Disj, and instead uses Cons and 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.

Theoretical Connection between BUA and FUA Triples. Note that while it is useful to have both FUA and BUA triples, *theoretically* speaking, only the BUA triples are needed for proving non-termination. As such, the BUA proof system constitutes one of our main contributions (while the FUA proof system was previously developed by de Vries and Koutavas [2011]; O'Hearn [2019]. Moreover, as mentioned above, with the exception of their associated rules of consequence (ConsF and ConsB in Fig. 1) all other FUA and BUA reasoning principles and proof rules coincide. In §5 we bolster this intuition (Theorem 10) by showing that given any under-approximate triple $[p]$ C $[\epsilon : q]$, if $[p]$ C $[\epsilon : q]$ is a valid *FUA* triple *and* its pre-states (p) are *FUA-minimal*, then $[p]$ C $[\epsilon : q]$ 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 $[\epsilon : q]$ is not a valid FUA triple. Intuitively, this ensures that pre-states p have not been arbitrarily weakened (grown) using ConsF.

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

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 an existing under-approximate tool such as Pulse with divergence proof rules. In particular, the divergence rules in Fig. 2 (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. 2); or 2) inductive cases, where the premises contain other divergent triples (e.g. Div-Seq1 in Fig. 2) or a combination of divergent and BUA triples (e.g. Div-Seq2 in Fig. 2).

For the base cases such as LoopFix, thanks to the forward reachability of BUA triples, we already establish 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.

3 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) [Ishtiaq and O'Hearn 2001] and incorrectness separation logic (ISL) [Raad et al. 2020]. Later in §6 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 ($\mathsf{assume}(B)$), scoped variable declaration ($\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}$), sequential composition ($\mathsf{C}_1; \mathsf{C}_2$), non-deterministic choice ($\mathsf{C}_1 + \mathsf{C}_2$) and loops ($\mathsf{C}^\star$), as well as explicit error statements (error, which can be thought of e.g. as $\mathsf{assert}(\mathsf{false})$).

$$\mathsf{C} ::= \mathsf{skip} \mid x := e \mid \mathsf{assume}(B) \mid \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C} \mid \mathsf{error} \mid \mathsf{C}_1 + \mathsf{C}_2 \mid \mathsf{C}_1; \mathsf{C}_2 \mid \mathsf{C}^\star$$

As is standard, deterministic choice and loops can be encoded using their non-deterministic counterparts and assume statements. Specifically, $\mathsf{if}\ (B)\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2$ can be encoded as $(\mathsf{assume}(B); \mathsf{C}_1) + (\mathsf{assume}(\neg B); \mathsf{C}_2)$, and $\mathsf{while}\ (B)\ \mathsf{C}$ can be encoded as $(\mathsf{assume}(B); \mathsf{C})^\star; \mathsf{assume}(\neg B)$.

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. $\neg p \triangleq p \Rightarrow \mathsf{false}$). We use p, q, r and their variants (e.g. p') as metavariables for assertions. An assertion describes a set of states, where each state is a (variable) store in $\textsc{Store} \triangleq \textsc{Var} \rightarrow \textsc{Val}$, mapping program variables to values.

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

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 $\emptyset$, $p \vee q$ for $p \cup q$, $p \Rightarrow q$ for state set inclusion ($p \subseteq q$), and so forth. Similarly, $e \oplus e'$ denotes sets of states (stores) in which $s(e) \oplus s(e')$ holds. As discussed in §2, we introduce the notion of *indexed disjunctions*, $P, Q \in \mathbb{N} \xrightarrow{\text{fin}} \mathcal{P}(\textsc{State})$, as a map from numbers to assertions (disjuncts); i.e. $P \equiv \bigvee_{i \in dom(P)} P(i)$.

UNTer Under-Approximate Proof Rules for Termination. Recall from §2 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. 1. The rules denoted by $\vdash_\dagger$ are *FUA and BUA* rules in that they are valid when interpreted in either the forward ($\vdash_\mathrm{F}$) or backward ($\vdash_\mathrm{B}$) direction. Note that as discussed in §2, with the exception of ConsF and ConsB rules, all rules in Fig. 1 are valid FUA *and* BUA triples.

$$\textsc{Skip}\qquad \vdash_\dagger [p]\,\mathsf{skip}\,[ok\!:p]$$

$$\textsc{Assign}\qquad \frac{y \notin \mathsf{fv}(p)}{\vdash_\dagger [p]\, x := e\, [ok\!: \exists y.\, p[y/x] \wedge x = e[y/x]]}$$

$$\textsc{Assume}\qquad \vdash_\dagger [p \wedge B]\ \mathsf{assume}(B)\ [ok\!: p \wedge B]$$

$$\textsc{Error}\qquad \vdash_\dagger [p]\ \mathsf{error}\ [er\!: p]$$

$$\textsc{Seq}\qquad \frac{\vdash_\dagger [p]\ \mathsf{C}_1\ [ok\!: r] \qquad \vdash_\dagger [r]\ \mathsf{C}_2\ [\epsilon : q]}{\vdash_\dagger [p]\ \mathsf{C}_1; \mathsf{C}_2\ [\epsilon : q]}$$

$$\textsc{SeqEr}\qquad \frac{\vdash_\dagger [p]\ \mathsf{C}_1\ [er\!: q]}{\vdash_\dagger [p]\ \mathsf{C}_1; \mathsf{C}_2\ [er\!: q]}$$

$$\textsc{Choice}\qquad \frac{\vdash_\dagger [p]\ \mathsf{C}_i\ [\epsilon : q] \qquad \text{for some } i \in \{1, 2\}}{\vdash_\dagger [p]\ \mathsf{C}_1 + \mathsf{C}_2\ [\epsilon : q]}$$

$$\textsc{Loop0}\qquad \vdash_\dagger [p]\ \mathsf{C}^\star\ [ok\!: p]$$

$$\textsc{Loop}\qquad \frac{\vdash_\dagger [p]\ \mathsf{C}^\star; \mathsf{C}\ [\epsilon : q]}{\vdash_\dagger [p]\ \mathsf{C}^\star\ [\epsilon : q]}$$

$$\textsc{Loop-Subvar}\qquad \frac{\forall n < k.\ \vdash_\dagger [p(n)]\,\mathsf{C}\,[ok\!:p(n{+}1)]}{\vdash_\dagger [p(0)]\ \mathsf{C}^\star\ [ok\!: p(k)]}$$

$$\textsc{Local}\qquad \frac{\vdash_\dagger [p]\ \mathsf{C}\ [\epsilon : q]}{\vdash_\dagger [\exists x.\, p]\,\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\,[\epsilon : \exists x.\, q]}$$

$$\textsc{Subst}\qquad \frac{\vdash_\dagger [p]\,\mathsf{C}\,[\epsilon : q] \quad x \notin \mathsf{fv}(p, \mathsf{C}, q)}{(\vdash_\dagger [p]\ \mathsf{C}\ [\epsilon : q])[y/x]}$$

$$\textsc{Disj}\qquad \frac{\vdash_\dagger [p_i]\ \mathsf{C}\ [\epsilon : q_i] \quad \text{for all } i \in I}{\vdash_\dagger \left[\bigvee_{i\in I} p_i\right]\ \mathsf{C}\ \left[\epsilon : \bigvee_{i\in I} q_i\right]}$$

$$\textsc{Constancy}\qquad \frac{\vdash_\dagger [p]\ \mathsf{C}\ [\epsilon : q] \quad \mathsf{fv}(r) \cap \mathsf{mod}(\mathsf{C}) = \emptyset}{\vdash_\dagger [p \wedge r]\ \mathsf{C}\ [\epsilon : q \wedge r]}$$

$$\textsc{ConsF}\qquad \frac{p' \subseteq p \quad \vdash_{\mathsf{F}} [p']\ \mathsf{C}\ [\epsilon : q'] \quad q \subseteq q'}{\vdash_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]}$$

$$\textsc{ConsB}\qquad \frac{p \subseteq p' \quad \vdash_{\mathsf{B}} [p']\ \mathsf{C}\ [\epsilon : q'] \quad q' \subseteq q}{\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]}$$

$$\textsc{DisjTrack}\qquad \frac{\vdash_\dagger [P_1]\ \mathsf{C}\ [\epsilon : Q_1] \quad \vdash_\dagger [P_2]\ \mathsf{C}\ [\epsilon : Q_2]}{\vdash_\dagger [P_1 \uplus P_2]\ \mathsf{C}\ [\epsilon : Q_1 \uplus Q_2]}$$

$$\textsc{Cons}\qquad \frac{\vdash_\dagger [P]\ \mathsf{C}\ [\epsilon : Q] \quad I \subseteq dom(P)}{\vdash_\dagger [P \downarrow I]\ \mathsf{C}\ [\epsilon : Q \downarrow I]}$$

$$\textsc{IfTrue}\qquad \frac{\vdash_\dagger [p \wedge B]\ \mathsf{C}_1\ [\epsilon : q]}{\vdash_\dagger [p \wedge B]\ \mathsf{if}\ (B)\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2\ [\epsilon : q]}$$

$$\textsc{IfFalse}\qquad \frac{\vdash_\dagger [p \wedge \neg B]\ \mathsf{C}_2\ [\epsilon : q]}{\vdash_\dagger [p \wedge \neg B]\ \mathsf{if}\ (B)\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2\ [\epsilon : q]}$$

$$\textsc{ConsEq}\qquad \frac{p \Leftrightarrow p' \quad \vdash_\dagger [p']\ \mathsf{C}\ [\epsilon : q'] \quad q' \Leftrightarrow q}{\vdash_\dagger [p]\ \mathsf{C}\ [\epsilon : q]}$$

$$\textsc{WhileFalse}\qquad \vdash_\dagger [p \wedge \neg B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [ok\!: p \wedge \neg B]$$

$$\textsc{WhileSubvar}\qquad \frac{\forall n < k.\ \ \vdash_\dagger [p(n) \wedge B]\ \mathsf{C}\ [ok\!: p(n{+}1) \wedge B] \qquad \vdash_\dagger [p(k) \wedge B]\ \mathsf{C}\ [\epsilon : q \wedge \neg B]}{\vdash_\dagger [p(0) \wedge B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\epsilon : q \wedge \neg B]}$$

Fig. 1. Under-approximate proof rules where † in each rule can be instantiated as F or B; the highlighted rules can be derived from other rules (see §B).

The Skip, Error, Seq, SeqEr, Choice, Loop0, Loop and Disj rules are identical to those of existing FUA logics [O'Hearn 2019; Raad et al. 2020, 2022]. Specifically, executing skip and error leave the state unchanged (Skip and Error), where the former terminates normally while the latter terminates erroneously; Disj allows us to merge multiple triples into one in a lossy fashion (as discussed in §2); the behaviour of a branching program can be under-approximated as the behaviour of *some* of its branches (Choice); and the behaviour of a loop can be under-approximated through bounded unrolling as zero (Loop0) or more (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 Loop-Subvar: 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 $\mathsf{C}^\star$ (i.e. executing C for k iterations) from the initial states $p(0)$. The SeqEr captures the short-circuiting behaviour of erroneous executions: if C_1 terminates erroneously, then $\mathsf{C}_1; \mathsf{C}_2$ also terminates erroneously. By contrast, 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 ϵ (either ok or er) and transforms r to q, then executing $\mathsf{C}_1; \mathsf{C}_2$ terminates as ϵ, transforming p to q.

The Assign rule is identical to the standard Floyd assignment rule and holds for both FUA and BUA. Observe that as noted by O'Hearn [2019], the Hoare assignment rule is not sound for FUA. That is, $\vdash_F \left[p[e/x]\right]\ x := e\ \left[ok\colon p\right]$ 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. $\vdash_B \left[p[e/x]\right]\ x := e\ \left[ok\colon p\right]$ is a sound BUA triple. However, this difference between BUA and FUA does not have a practical ramification as the Floyds assignment rule (in Assign) is sufficient to enable automated reasoning in Pulse.

The Assume, Local and Constancy rules are analogous to the FUA rules of [O'Hearn 2019]. Concretely, executing $\mathsf{assume}(B)$ terminates normally and leaves the state unchanged, provided that B holds beforehand. When executing the scoped variable declaration $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}$, the information about x is erased by existentially quantifying it in the pre- and post-states. The Constancy rule is used to adapt triples in different contexts and states: if an assertion r holds before executing C, it also holds afterwards provided that it does not refer to free variables that may have been modified by C. This is captured by the $\mathsf{fv}(r) \cap \mathsf{mod}(\mathsf{C}) = \emptyset$, where $\mathsf{fv}(r)$ denotes the free variables of r and $\mathsf{mod}(\mathsf{C})$ denotes the variables modified by C (i.e. those on the left-hand side of assignments).

As discussed in §2, 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 $dom(P \downarrow I) = I$ 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 §B). The IfTrue (resp. IfFalse) is analogous to its non-deterministic counterpart (Choice) and requires that condition B hold (resp. not hold) at the beginning. The ConsEq simply replaces implication (subset inclusion) in the premises of ConsF and ConsB with equivalence. The WhileFalse states that the pre-states are unchanged by the loop if the condition B does not hold to begin with (i.e. the loop is never entered). The WhileSubvar is analogous to Loop-Subvar and states that if for all $n < k$ an execution of C transforms $p(n) \wedge B$ to $p(n{+}1) \wedge B$, i.e. loop condition B remains true in the first $k{-}1$ iterations, and the k^{th} iteration results in the states in $q \wedge \neg B$ (i.e. it invalidates the loop condition), then $\mathsf{while}\ (B)\ \mathsf{C}$ terminates, transforming the initial states in $p(0) \wedge B$ to those in $q \wedge \neg B$.

UNTer Divergent Proof Rules for Non-Termination. We present the (syntactic) proof rules for divergence in Fig. 2. Recall from §2 that $[p]\ \mathsf{C}\ [\infty]$ states that every state in p leads to *some* divergent trace. We provide the formal semantic interpretation of divergent triples later in §5.

Note that skip, assignment, error and assume statements never diverge. In order to show that $\mathsf{C}_1; \mathsf{C}_2$ 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 $\mathsf{C}_1 + \mathsf{C}_2$ has a divergent trace starting from p, it suffices to show that *some* branch C_i has a divergent trace from p, i.e. in an

$$\textsc{Div-Seq1}\quad \frac{\vdash [p]\ \mathsf{C}_1\ [\infty]}{\vdash [p]\ \mathsf{C}_1;\mathsf{C}_2\ [\infty]}
\qquad
\textsc{Div-Seq2}\quad \frac{\vdash_{\mathsf{B}} [p]\ \mathsf{C}_1\ [ok\colon q] \qquad \vdash [q]\ \mathsf{C}_2\ [\infty]}{\vdash [p]\ \mathsf{C}_1;\mathsf{C}_2\ [\infty]}
\qquad
\textsc{Div-Choice}\quad \frac{\vdash [p]\ \mathsf{C}_i\ [\infty] \quad \text{for some } i \in \{1,2\}}{\vdash [p]\ \mathsf{C}_1 + \mathsf{C}_2\ [\infty]}$$

$$\textsc{Div-LoopUnfold}\quad \frac{\vdash [p]\ \mathsf{C};\mathsf{C}^\star\ [\infty]}{\vdash [p]\ \mathsf{C}^\star\ [\infty]}
\qquad
\textsc{Div-Loop}\quad \frac{\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [ok\colon q] \qquad q \subseteq p}{\vdash [p]\ \mathsf{C}^\star\ [\infty]}
\qquad
\textsc{Div-SubVar}\quad \frac{\forall n \in \mathbb{N}.\ \vdash_{\mathsf{B}} [p(n)]\ \mathsf{C}\ [ok\colon p(n{+}1)]}{\vdash [p(0)]\ \mathsf{C}^\star\ [\infty]}$$

$$\textsc{Div-Local}\quad \frac{\vdash [p]\ \mathsf{C}\ [\infty]}{\vdash [\exists x.\, p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\infty]}
\qquad
\textsc{Div-Subst}\quad \frac{\vdash [p]\ \mathsf{C}\ [\infty] \qquad y \notin \mathsf{fv}(p,\mathsf{C})}{\vdash ([p]\ \mathsf{C}\ [\infty])[y/x]}
\qquad
\textsc{Div-Cons}\quad \frac{\vdash [p']\ \mathsf{C}\ [\infty] \qquad p \subseteq p'}{\vdash [p]\ \mathsf{C}\ [\infty]}$$

$$\textsc{Div-Disj}\quad \frac{\vdash [p_i]\,\mathsf{C}\,[\infty] \quad \text{for all } i \in I}{\vdash \left[\bigvee_{i \in I} p_i\right]\ \mathsf{C}\ [\infty]}
\qquad
\textsc{Div-LoopNest}\quad \frac{\vdash [p]\ \mathsf{C}\ [\infty]}{\vdash [p]\,\mathsf{C}^\star\,[\infty]}
\qquad
\textsc{Div-While}\quad \frac{\vdash_{\mathsf{B}} [p \wedge B]\,\mathsf{C}\,[ok\colon q \wedge B] \qquad q \subseteq p}{\vdash [p \wedge B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\infty]}$$

$$\textsc{Div-WhileNest}\quad \frac{\vdash [p \wedge B]\ \mathsf{C}\ [\infty]}{\vdash [p \wedge B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\infty]}
\qquad
\textsc{Div-WhileSubVar}\quad \frac{\forall n \in \mathbb{N}.\ \vdash_{\mathsf{B}} [p(n) \wedge B]\,\mathsf{C}\,[ok\colon p(n{+}1) \wedge B]}{\vdash [p(0) \wedge B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\infty]}$$

Fig. 2. The UNTer divergence rules, where the highlighted rules can be derived from other rules

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 Div-Local rule states that variable declaration diverges if its body does. The Div-Disj rule denotes that if the states in each of p_i lead to divergence, then so do the states in their union. The Div-Subst rule is the substitution rule for divergence and is as expected.

The remaining rules capture divergence for loops. Specifically, 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.

$$\cfrac{\cfrac{\cfrac{}{[p]\ \mathsf{C}\ [\infty]}\ \text{(given)}}{[p]\ \mathsf{C};\mathsf{C}^\star\ [\infty]}\ (\textsc{Div-Seq1})}{[p]\ \mathsf{C}^\star\ [\infty]}\ (\textsc{Div-LoopUnfold})$$

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. $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [ok\colon q]$) and $q \subseteq p$, then $\mathsf{C}^\star$ has a divergent trace starting from p. Intuitively, the forward triple in the premise, $A \triangleq \vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [ok\colon q]$, allows us to construct an infinite trace of $\mathsf{C}^\star$ from any state in p: given a state in $s_0 \in p$, (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-SubVar 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 $\mathsf{C}^\star$ 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 $\mathsf{C}^\star$ always has a divergent trace. However, this is not necessarily the case with conditional if/while statements (encoded via assume). This is illustrated in Div-While, requiring that the loop condition

while $(x = 0)$
  skip
	while $(x \geq 0)$
	  $x := x{+}1$
		$x := 1$
		$y := 2$;
		while $(x{+}y > 1)$
		  $x := 3 - x$
		  $y := 3 - y$
			while $(y < 100)$
			  if $(y \leq 50)$
			    $x := x{+}1$
			  else
			    $y := y{+}1$
				while $(y < 100)$
				  $x := 0$;
				  while $(x \leq 100)$
				    if $(x = 100)$
				      $y := 0$
				    $x := x{+}1$
				  $y := y{+}1$
					$x := 42$; $y := 1$;
					while $(y < 100)$
					  while $(x \leq 100)$
					    if $(x = 100)$
					      $x := 1$
					      $y := 2 \times y$
					    else $x := x{+}1$
					  $y := y{+}1$

(a)	(b)	(c)	(d)	(e)	(f)

Fig. 3. Several examples of programs with non-terminating behaviours where x, y initially hold 0

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 $x := 1$. The Div-WhileNest and Div-WhileSubvar rules are analogous to Div-LoopNest and Div-Subvar, respectively. As before, all highlighted rules in Fig. 2 can be derived from other rules (see §B).

4 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$^\infty$ prototype (see §7).

Example 1 (Fig. 3a). Consider the simple example in Fig. 3a comprising a simple divergent loop. We can detect this using Div-While (with $p = q = \text{true}$) as shown below:

$$\dfrac{\dfrac{}{\vdash_{\mathrm{B}} [x = 0]\ \text{skip}\ [ok\colon x = 0]}\ (\textsc{Skip})}{[x = 0]\ \text{while}\ (x = 0)\ \text{skip}\ [\infty]}\ (\textsc{Div-While})$$

Example 2 (Fig. 3b). Consider the simple example in Fig. 3b comprising a simple while loop with a buggy check. We can detect this using Div-While (with $p = \text{true}$ and $q = x \geq 1$) as shown below:

$$\dfrac{\dfrac{\dfrac{}{\vdash_{\mathrm{B}} [x \geq 0]\ x := x{+}1\ [ok\colon \exists v.\ v \geq 0 \land x = v{+}1]}\ (\textsc{Assign})}{\vdash_{\mathrm{B}} [x \geq 0]\ x := x{+}1\ [ok\colon x \geq 1 \land x \geq 0]}\ (\textsc{ConsEq}) \qquad \dfrac{}{x \geq 1 \subseteq x \geq 0}}{[x \geq 0]\ \text{while}\ (x \geq 0)\ x := x{+}1\ [\infty]}\ (\textsc{Div-While})$$

Example 3 (Fig. 3c). Consider the example in Fig. 3c. 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. 4. 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. 3d). Consider the example in Fig. 3d. 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

```
1. $[x = 0 \land y = 0]$
2.    $x := 1$; // Assign, ConsEq
3. $[ok: x = 1 \land y = 0]$
4.    $y := 2$; // Assign, ConsEq
5. $[ok: x = 1 \land y = 2]$ // Div-Cons
6. $[ok: x{+}y = 3 \land x{+}y > 1]$
7.    while $(x{+}y > 1)$
   Div-While
      8. $[x{+}y = 3 \land x{+}y > 1]$
      9.    $x := 3 - x$ // Assign
      10. $\left[ok: \begin{array}{l} \exists v_x.\ v_x{+}y = 3 \land v_x{+}y > 1 \\ \land\ x = 3 - v_x \end{array}\right]$
      11.    $y := 3 - y$ // Assign
      12. $\left[ok: \begin{array}{l} \exists v_x, v_y.\ v_x{+}v_y = 3 \land v_x{+}v_y > 1 \\ \land\ x = 3 - v_x \land y = 3 - v_y \end{array}\right]$
      // ConsEq
      13. $[ok: x{+}y = 3 \land x{+}y > 1]$
14. $[\infty]$
```

```
1. $[x = 0 \land y = 0]$
// Div-Cons
2. $[y = 0 \land y < 100]$
3.    while $(y < 100)$
   Div-While
      4. $[y = 0 \land y < 100]$
      // ConsEq
      5. $[y = 0 \land y < 100 \land y \leq 50]$
      6.    if $(y \leq 50)$
         IfTrue
            7. $[y = 0 \land y < 100 \land y \leq 50]$
            8.    $x := x{+}1$ // Assign
            9. $\left[ok: \begin{array}{l} \exists v_x.\ y = 0 \land y < 100 \\ \land\ y \leq 50 \land x = v_x{+}1 \end{array}\right]$
            // ConsEq
            10. $[ok: y = 0 \land y < 100]$
         11.    else $\cdots$
      12. $[ok: y = 0 \land y < 100]$
13. $[\infty]$
```

Fig. 4. Proof sketches of the divergence bugs in Fig. 3c (left) and Fig. 3d (right)

right of Fig. 4. After applying ConsEq to rewrite p equivalently as $p \land y \leq 50$ (line 5), we apply IfTrue to show we can take the then branch and arrive at p (lines 7–10).

Example 5 (Fig. 3e). Consider the example in Fig. 3e 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. 5. 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. 5. We then apply WhileSubvar to show that at the end of the execution of the inner loop we arrive at $y{=}0 \land x{=}101 \land x \nleq 100$ (lines 7–21). Note that WhileSubvar 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 $p(n) \land x \leq 100$ to $p(n{+}1) \land x \leq 100$; 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 $y{=}0 \land x{=}101 \land x \nleq 100$ which is included in $y < 100$ (line 22), as per the second premise of Div-While.

Example 6 (Fig. 3f). Consider the nested loops in Fig. 3f. 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. 5. 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 Div-Cons to weaken the states to $x \leq 100$ (line 7) and subsequently apply Div-While (lines 7–20) to show that the body of the inner loop leaves the states $x \leq 100$ unchanged (lines 9–19). To do this, we first rewrite $x \leq 100$ equivalently as $x < 100 \lor x = 100$ (line 10), and then apply Disj to show that either disjunct results in $x \leq 100$ states (the two columns on lines 11–14 and 15–18). The proof of each disjunct is then straightforward and is obtained by reasoning about the associated branch.

```
1. $[x = 0 \land y = 0]$ // Div-Cons
2. $[y < 100]$
3.   while $(y < 100)$
   Div-While
      4. $[y < 100]$
      5.    $x := 0$ // Assign
      6. $[ok\colon y < 100 \land x = 0]$ // ConsEq
      7. $[ok\colon p(0) \land x \leq 100]$
      8.    while $(x \leq 100)$
         WhileSubvar
            9. $\forall n < 100.\ [p(n) \land n{<}100 \land x \leq 100]$
            10.        if $(x = 100)\ y := 0$
            11.        else skip // IfFalse, Skip
            12.    $[ok\colon p(n) \land n{<}100 \land x{\leq}100]$
            13.        $x := x{+}1$ // Assign, ConsEq
            14.    $[ok\colon p(n{+}1) \land x \leq 100]$

            15. $[p(100) \land x \leq 100]$
            16.    if $(x = 100)\ y := 0$
            17.    else skip // IfTrue, Assign
            18. $[ok\colon p(100) \land x \leq 100 \land y = 0]$
            19.    $x := x{+}1$ // Assign, ConsEq
            20. $[ok\colon y = 0 \land x = 101 \land x \not\leq 100]$
     21. $[ok\colon y = 0 \land x = 101 \land x \not\leq 100]$
     22. $[ok\colon y < 100]$
23. $[\infty]$
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. $[ok\colon x = 42 \land y = 1]$ // Div-Cons
4. $[ok\colon x \leq 100 \land y < 100]$
5.   while $(y < 100)$
   Div-WhileNest
      6. $[x \leq 100 \land y < 100]$ // Div-Cons
      7. $[x \leq 100]$
      8.   while $(x \leq 100)$
         Div-While
            9. $[x \leq 100]$ // ConsEq
            10. $[x < 100 \lor x = 100]$
               Disj
                  11. $[x < 100]$
                  12.    if $(x = 100)\ x := 1;\ y := 2{\times}y$
                  13.    else $x := x{+}1$
                         // IfFalse, Assign, ConsEq
                  14. $[ok\colon x \leq 100]$

                  15. $[x = 100]$
                  16.    if $(x = 100)\ x := 1;\ y := 2{\times}y$
                  17.    else $x := x{+}1$
                         // IfTrue, Assign, ConsB
                  18. $[ok\colon x \leq 100]$
            19. $[ok\colon x \leq 100]$
      20. $[\infty]$
21. $[\infty]$
```

Fig. 5. Proof sketch of divergence in Fig. 3e (above), where the two columns on lines 9–14 and 15–20 denote the proof sketches of the two premises of WhileSubvar; proof sketch of divergence in Fig. 3f (below), where the two columns on lines 11–14 and 15–18 denote the proof sketches of the two premises of Disj.

5 The UNTer Model and Semantics

Instrumented Commands and Operational Semantics. Although in sequential settings the semantics is given in the big-step fashion [O'Hearn 2019; Raad et al. 2020], 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 §3), which additionally include the end(x, v) construct, recording the existing (old) value of x when redeclaring x in a new scope.

$$\mathbb{C} ::= \mathsf{C} \mid \mathsf{end}(x, v) \mid \mathbb{C}_1; \mathbb{C}_2$$

$$\textsc{S-Local}\ \frac{s' = s[x \mapsto v] \quad v \in \textsc{Val}}{\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, s \to \mathsf{C}; \mathsf{end}(x, s(x)), s', ok} \qquad \textsc{S-LocalEnd}\ \frac{s' = s[x \mapsto v]}{\mathsf{end}(x, v), s \to \mathsf{skip}, s', ok} \qquad \textsc{S-Assign}\ \frac{s' = s[x \mapsto s(e)]}{x := e, s \to \mathsf{skip}, s', ok}$$

$$\textsc{S-Assume}\ \frac{s(B) = \mathsf{true}}{\mathsf{assume}(B), s \to \mathsf{skip}, s, ok} \qquad \textsc{S-Error}\ \mathsf{error}, s \to \mathsf{skip}, s, er \qquad \textsc{S-Choice}\ \frac{i \in \{1, 2\}}{\mathsf{C}_1 + \mathsf{C}_2, s \to \mathsf{C}_i, s, ok} \qquad \textsc{S-Seq1}\ \frac{\mathbb{C}_1, s \to \mathbb{C}_1', s', \epsilon}{\mathbb{C}_1; \mathbb{C}_2, s \to \mathbb{C}_1'; \mathbb{C}_2, s', \epsilon}$$

$$\textsc{S-SeqSkip}\ \mathsf{skip}; \mathbb{C}, s \to \mathbb{C}, s, ok \qquad \textsc{S-Loop0}\ \mathsf{C}^\star, s \to \mathsf{skip}, s, ok \qquad \textsc{S-Loop}\ \mathsf{C}^\star, s \to \mathsf{C}; \mathsf{C}^\star, s, ok$$

$$\frac{\mathbb{C} \in \{\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, x := e, \mathsf{assume}(B), \mathsf{error}, \mathsf{C}_1 + \mathsf{C}_2, \mathsf{C}^\star\}}{\mathbb{C}, s \rightsquigarrow_{\mathrm{er}} \mathsf{skip}, s} \qquad \frac{\mathbb{C}_1, s \rightsquigarrow_{\mathrm{er}} \mathbb{C}_1', s'}{\mathbb{C}_1; \mathbb{C}_2, s \rightsquigarrow_{\mathrm{er}} \mathbb{C}_1'; \mathbb{C}_2, s'}$$

$$\mathsf{end}(x, v), s \rightsquigarrow_{\mathrm{er}} \mathsf{skip}, s[x \mapsto v] \qquad \mathsf{skip}; \mathbb{C}, s \rightsquigarrow_{\mathrm{er}} \mathbb{C}, s$$

Fig. 6. The UNTer small-step transitions (above) and error transitions for restoring variables (below)

We present our small-step semantics in Fig. 6, with transitions of the form $\mathbb{C}, s \to \mathbb{C}', s', \epsilon$, where $\mathbb{C}$ and s respectively denote the current (instrumented) command and store (state), $\mathbb{C}'$ and s' denote their continuations (what they reduce to) and ϵ denotes the exit condition, describing whether reducing $\mathbb{C}$ to $\mathbb{C}'$ took place normally (ok) or erroneously (er). As shown in S-Local, when evaluating $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}$ under a state $s \in \textsc{Store}$, we assign an arbitrary value v to x in s, and continue with executing C followed by $\mathsf{end}(x, s(x))$. 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; $\mathsf{assume}(B)$ reduces to skip normally when B evaluates to true in the current state; error reduces to skip erroneously; and $\mathsf{C}_1 + \mathsf{C}_2$ non-deterministically reduces to one of its branches (C_i with $i \in \{1, 2\}$). When reducing $\mathbb{C}_1; \mathbb{C}_2$, 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 $\mathsf{C}^\star$ (S-Loop).

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

Definition 1 (Semantic BUA and FUA triples)**.** A BUA triple is *valid*, written $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$, iff for all $s_p \in p$, there exists $s_q \in q$ and n such that $\mathsf{C}, s_p \xrightarrow{n} -, s_q, \epsilon$, where:

$$\begin{aligned} \mathbb{C}, s \xrightarrow{n} \mathbb{C}', s', \epsilon \stackrel{\text{def}}{\iff} & (n{=}0 \land \mathbb{C}{=}\mathbb{C}'{=}\mathsf{skip} \land s{=}s' \land \epsilon{=}ok) \\ & \lor (n{=}1 \land \epsilon \in \textsc{ErExit} \land \exists s''.\ \mathbb{C}, s \to \mathbb{C}', s'', \epsilon \land \mathbb{C}', s'' \rightsquigarrow_{\mathrm{er}}^{+} \mathsf{skip}, s') \\ & \lor (\exists k, \mathbb{C}'', s''.\ n{=}k{+}1 \land \mathbb{C}, s \to \mathbb{C}'', s'', ok \land \mathbb{C}'', s'' \xrightarrow{k} \mathbb{C}', s', \epsilon) \end{aligned}$$

and $\mathbb{C}, s \to \mathbb{C}', s', \epsilon$ is the UNTer small-step transitions given at the top of Fig. 6, while $\rightsquigarrow_{\mathrm{er}}^{+}$ denotes the transitive closure of the error transitions $\rightsquigarrow_{\mathrm{er}}$ as defined at the bottom of Fig. 6 (described shortly). A FUA triple is *valid*, written $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$, iff for all $s_q \in q$, there exists $s_p \in p$ and n such that $\mathsf{C}, s_p \xrightarrow{n} -, s_q, \epsilon$.

The first disjunct in $\mathbb{C}, s \xrightarrow{n} \mathbb{C}', s', \epsilon$ denotes that a state is reached under ok in zero steps without changing the underlying state, provided that $\mathbb{C}$ is simply skip. The last disjunct captures the inductive case (n=k+1), where $\mathbb{C}$ takes an ok step, and s' is subsequently reached in k steps under ϵ.

Finally, the second disjunct captures the short-circuiting semantics of errors: a state s' is reached in one step under er when $\mathbb{C}$ takes an erroneous step, whereupon locally declared variables (through local) are restored to their oldest values (outer-most scope) via $\rightsquigarrow_{\mathsf{er}}$ transitions (defined in Fig. 6). The $\rightsquigarrow_{\mathsf{er}}$ transitions 'skip over' the execution of most commands and restore the value of a variable x when encountering $\mathsf{end}(x, v)$. Specifically, the $\rightsquigarrow_{\mathsf{er}}$ transitions of all commands, except those of $\mathsf{end}(x, v)$ and sequential composition, do not change the underlying state and simply reduce to skip (i.e. ignore their effects), while the $\rightsquigarrow_{\mathsf{er}}$ transition for $\mathsf{end}(x, v)$ restores the value of x to v. The $\rightsquigarrow_{\mathsf{er}}$ transitions for sequential composition are defined inductively as expected.

We next show that the BUA and FUA proof systems presented in Fig. 1 are *both sound and complete*, with the full proof given in §C.1 and §D.1.

Theorem 7 (BUA and FUA soundness). *For all p, q, C and ϵ:*

1) if $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 1, then $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds; and
2) if $\vdash_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 1, then $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds.

Theorem 8 (BUA and FUA completeness). *For all p, q, C and ϵ:*

1) if $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds, then $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 1; and
2) if $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds, then $\vdash_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 1.

Definition 2 (Semantic divergent triples). A divergent triple is *valid*, written $\models [p]\ \mathsf{C}\ [\infty]$, iff for all $s \in p$, there exists an infinite series of $\mathbb{C}_1, \mathbb{C}_2, \cdots$, $s_1, s_2, \cdots$ and $n_1, n_2, \cdots$ such that $\mathsf{C}, s \rightsquigarrow^{n_1} \mathbb{C}_1, s_1, ok \rightsquigarrow^{n_2} \mathbb{C}_2, s_2, ok \rightsquigarrow^{n_3} \cdots$, where the chain $\mathsf{C}, s \rightsquigarrow^{n_1} \mathbb{C}_1, s_1, ok \rightsquigarrow^{n_2} \mathbb{C}_2, s_2, ok \rightsquigarrow^{n_3} \cdots$ is a shorthand for $\mathsf{C}, s \rightsquigarrow^{n_1} \mathbb{C}_1, s_1, ok \land \mathbb{C}_1, s_1 \rightsquigarrow^{n_2} \mathbb{C}_2, s_2, ok \land \cdots$, and $\rightsquigarrow^{n}$ is defined as follows:

$$\begin{array}{lcl} \mathbb{C}, s \rightsquigarrow^{n} \mathbb{C}', s', \epsilon & \overset{\text{def}}{\iff} & (n = 1 \land \epsilon = ok \land \mathbb{C}, s \to \mathbb{C}', s', \epsilon) \\ & & \lor\ (n = 1 \land \epsilon \in \text{\textsc{ErExit}} \land \exists s''.\ \mathbb{C}, s \to \mathbb{C}', s'', \epsilon \land \mathbb{C}', s'' \rightsquigarrow^{+}_{\mathsf{er}} \mathsf{skip}, s') \\ & & \lor\ (\exists k, s'', \mathbb{C}''.\ n{=}k{+}1 \land \mathbb{C}, s \to \mathbb{C}'', s'', ok \land \mathbb{C}'', s'' \rightsquigarrow^{k} \mathbb{C}', s', \epsilon) \end{array}$$

Note that unlike the $\mathbb{C}, s \xrightarrow{n} \mathbb{C}', s'$ transitions in Def. 1 which describe *terminating* traces (by reduction to skip), the $\mathbb{C}, s \rightsquigarrow^{n} \mathbb{C}', s'$ transitions do not stipulate termination and simply state that executing $\mathbb{C}$ from s for n steps reduces to $\mathbb{C}'$ and results in s'.

We next show that the divergence proof system presented in Fig. 2 is *both sound and complete*, with the full proof given in §C.2 and §D.2.

Theorem 9 (Divergence soundness and completeness). *For all p and C, if $\vdash [p]\ \mathsf{C}\ [\infty]$ is derivable using the rules in Fig. 2, then $\models [p]\ \mathsf{C}\ [\infty]$ holds.*
For all p and C, if $\models [p]\ \mathsf{C}\ [\infty]$ holds, then $\vdash [p]\ \mathsf{C}\ [\infty]$ is derivable using the rules in Fig. 2.

Finally, we formalise the relationship between FUA and BUA triples (see p. 8), with the proof in §E.

Theorem 10. *For all p, C, q, ϵ:*

1) if $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ and $\mathsf{min}_{\mathsf{pre}}(p, \mathsf{C}, q)$ hold, then $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ also holds; and
2) if $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ and $\mathsf{min}_{\mathsf{post}}(p, \mathsf{C}, q)$ hold, then $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ also holds, where:

$$\mathsf{min}_{\mathsf{pre}}(p, \mathsf{C}, q) \overset{def}{\iff} \forall p'.\ p' \subset p \Rightarrow \not\models_{\mathsf{F}} [p']\,\mathsf{C}\,[\epsilon : q] \qquad \mathsf{min}_{\mathsf{post}}(p, \mathsf{C}, q) \overset{def}{\iff} \forall q'.\ q' \subset q \Rightarrow \not\models_{\mathsf{B}} [p]\,\mathsf{C}\,[\epsilon : q']$$

AssignSL
$$\vdash_{\dagger} [x{=}x'] \; x := e \; [ok : x{=}e[x'/x]]$$

Store
$$\vdash_{\dagger} [x \mapsto e] \; [x] := y \; [ok : x \mapsto y]$$

StoreEr
$$\vdash_{\dagger} [x \not\mapsto] \; [x] := y \; [er : x \not\mapsto]$$

StoreNull
$$\vdash_{\dagger} [x{=}\mathsf{null}] \; [x] := y \; [er : x{=}\mathsf{null}]$$

Frame
$$\frac{\vdash_{\dagger} [p] \, \mathsf{C} \, [\epsilon : q] \qquad \mathsf{mod}(\mathsf{C}) \cap \mathsf{fv}(r){=}\emptyset}{\vdash_{\dagger} [p * r] \; \mathsf{C} \; [\epsilon : q * r]}$$

Div-Frame
$$\frac{\vdash \; [p] \; \mathsf{C} \; [\infty]}{\vdash \; [p * r] \; \mathsf{C} \; [\infty]}$$

Fig. 7. UNTer$^{\mathsf{SL}}$ proof rules (excerpt), where x and x' are distinct variables and † in each rule can be instantiated as F or B; see §F (Fig. 10) for the full set of UNTer$^{\mathsf{SL}}$ rules.

Note that while the theoretical result in Theorem 10 does not have an immediate practical impact, it nevertheless reconciles FUA and BUA reasoning and shows how we can use tools such as Pulse that are underpinned by FUA to detect non-termination. Specifically, $\mathsf{min}_{\mathsf{pre}}$ describes a minimal precondition that has not been arbitrarily weakened (grown) using the ConsF rule. Similarly, $\mathsf{min}_{\mathsf{post}}$ describes a minimal postcondition that has not been arbitrarily weakened using ConsB. As such, given a set S of FUA triples inferred by a FUA-based analysis tool such as Pulse, the triples in S can be soundly interpreted as BUA ones (and therefore used to prove divergence), provided that their preconditions have not been weakened using ConsF, which is indeed the case in Pulse.

6 Extension to Separation Logic

We describe how we develop UNTer$^{\mathsf{SL}}$ by extending UNTer with the compositional reasoning principles of separation logic (SL) [Ishtiaq and O'Hearn 2001]. Raad et al. [2020] have developed incorrectness separation logic (ISL) by extending the FUA-based incorrectness logic (IL) [O'Hearn 2019] with separation logic. We adopt the model of Raad et al. [2020] and show that it is also sound for BUA reasoning.

UNTer$^{\mathsf{SL}}$ Programming Language and Assertions. To account for operations that access the heap, in UNTer$^{\mathsf{SL}}$ we extend our programming language from §3 with the following heap-manipulating operations (below, left) for allocation ($x := \mathsf{alloc}()$), deallocation ($\mathsf{free}(x)$), reading from the heap (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.

$$\begin{array}{llll} \textsc{Comm} \ni \mathsf{C} ::= & \cdots \mid x := \mathsf{alloc}() \mid \mathsf{free}(x) & \qquad \textsc{Ast} \ni p, q, r ::= & \cdots \mid \mathsf{emp} \mid e \mapsto e' \\ & \mid x := [y] \mid [x] := y & & \mid e \not\mapsto \;\mid p * q \end{array}$$

The UNTer$^{\mathsf{SL}}$ assertions describe sets of *states*, where a state comprises a (variable) store and a heap. Existing UNTer assertions from §3 then simply describe states where the heap is empty and the store satisfies the assertion. The structural assertions above are those of ISL [Raad et al. 2020], which themselves are standard SL assertions [Ishtiaq and O'Hearn 2001] extended with $e \not\mapsto$. The $e \not\mapsto$ describes states where the heap comprises a single location at e containing the designated value $\bot$. In particular, whilst $e \mapsto e'$ states that location e is allocated (and contains value e'), $e \not\mapsto$ states that location e is *deallocated.*

UNTer$^{\mathsf{SL}}$ Proof Rules (Syntactic UNTer$^{\mathsf{SL}}$ Triples). We present an excerpt of the UNTer$^{\mathsf{SL}}$ proof rules in Fig. 7; see §F (Fig. 10) for the full set of rules. Note that all UNTer rules (both BUA and FUA) in Fig. 1, except Constancy and Assign, are also UNTer$^{\mathsf{SL}}$ rules and are omitted from Fig. 7. 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 UNTer), UNTer$^{\mathsf{SL}}$ 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.

```
1. $[input[0] \mapsto 0 * size > 0 * \mathit{off} = v * newoff = - * i = -]$
2.   $\mathit{off} := 0$; // AssignSL, Frame
3. $\lfloor ok\colon input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = - * i = - \rfloor$ // ConsEq
4. $\lfloor ok\colon input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = - * i = - * \mathit{off} < size \rfloor$
5.  while $(\mathit{off} < size)$
    Div-While
    6. $[input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = - * i = - * \mathit{off} < size]$
    7.   $newoff := [input[\mathit{off}]]$ // Load, Frame
    8. $\lfloor ok\colon input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = 0 * i = - * \mathit{off} < size \rfloor$
    9.   $i := \mathit{off}$ // AssignSL, Frame
    10. $\lfloor ok\colon input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = 0 * i = 0 * \mathit{off} < size \rfloor$ // ConsEq
    11. $\lfloor ok\colon input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = 0 * i = 0 * \mathit{off} < size * \neg(i < newoff) \rfloor$
    12.  while $(i < newoff)$ $\{\cdots; i{+}{+}\}$ // WhileFalse
    13. $\lfloor ok\colon input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = 0 * i = 0 * \mathit{off} < size * \neg(i < newoff) \rfloor$
    14.  $\mathit{off} := \mathit{off} + newoff$ // AssignSL, Frame, ConsEq
    15. $\lfloor ok\colon input[0] \mapsto 0 * size > 0 * \mathit{off} = 0 * newoff = 0 * i = 0 * \mathit{off} < size \rfloor$
16. $[\infty]$
```

Fig. 8. UNTer$^{\text{SL}}$ proof sketch of CVE-2023-34966 in the Samba library (see Example 11)

As in SL and ISL, the crux of UNTer$^{\text{SL}}$ lies in the Frame rule, allowing one to extend the pre- and post-states with disjoint resources in r, where $\mathsf{fv}(r)$ returns the set of free variables in r, and $\mathsf{mod}(\mathrm{C})$ returns the set of (program) variables modified by C (i.e. those on the left-hand of ':=' 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 UNTer$^{\text{SL}}$ divergent rules are those of UNTer in Fig. 2, except that the BUA UNTer triples in the premises (e.g. the first premise of Div-Seq2) are replaced with their UNTer$^{\text{SL}}$ counterparts. Additionally, we extend the framing principle to divergent triples as shown in Div-Frame: if C diverges starting from the states in p, then it also diverges starting from the states in $p * r$.

We next use UNTer$^{\text{SL}}$ to detect a known divergence bug in the Samba library, which has already been reported to the Common Vulnerabilities and Exposures (CVE) database as CVE-2023-34966.

Example 11 (Samba). The example in Fig. 8 is a stylised excerpt from the Samba library, where the body of `sl_unpack_loop` is repeated below. The excerpt shown reads chunks of data, where the size of each chunk is given by the corresponding entry in the *input* array (the size of the first chunk is stored in *input*[0], the size of the second in *input*[1] and so forth). The *off* records the offset at which next chunk to be read is stored and is initially set to zero (line 2). At each iteration of the outer while loop (lines 5–16), the size of the next chunk is read from *input* into *newoff* (line 7), and subsequently the offset is incremented by *newoff* (line 14). The inner while loop (line 12) then proceeds to read the data between *off* and *newoff* (elided here as $\cdots$) one unit at a time (incrementing i each time). Note that when $i = newoff = 0$, then this inner loop is never entered. Moreover, if $newoff = 0$, then the old offset is never incremented (i.e. the increment at line 14 is idempotent), and thus the loop never terminates. This is indeed the cause of divergence in this example, which has since been patched by simply adding a check at the beginning of the outer loop, ensuring that *newoff* is non-zero and returning an error value when that is the case. Using UNTer$^{\text{SL}}$ we can detect this bug as shown in Fig. 8.

UNTer$^{\text{SL}}$ Semantics and Soundness. For brevity, we present the UNTer$^{\text{SL}}$ model and semantics in the appendix (see §F). The formal interpretations of BUA, FUA and divergent triples in

Tool	**Term.**	**Non-term.**	**Non-det.**	**Heap**	**Auto.**	**UA/OA**	**Large Code / Libraries**
Terminator [Cook et al. 2006a,b]	✓	✗	✓	✗	✓	OA	✗
Mutant [Berdine et al. 2006]	✓	✗	✗	✓	✓	OA	✗
TNT [Gupta et al. 2008]	✗	✓	✗	✗	✓	OA	✗
KEY [Velroyen and Rümmer 2008]	✗	✓	✗	✗	✓	OA	✗
CPROVER [Kroening et al. 2010]	✓	✓	✓	✗	✓	UA-OA	✗
TRex [Harris et al. 2010]	✓	✓	✓	✗	✓	UA-OA	✗
T2 [Cook et al. 2013]	✓	✗	✓	✗	✓	OA	✗
Coop-T2 [Brockschmidt et al. 2013]	✓	✓	✓	✗	✓	UA-OA	✗
Caber [Brotherston and Gorogiannis 2014]	✓	✗	✓	✓	✓	OA	✗
CPP-INV [Larraz et al. 2014]	✗	✓	✓	✗	✓	OA	✗
HipTNT [Le et al. 2014]	✓	✓	✓	Partial	✗	OA	✗
HipTNT+ [Le et al. 2015]	✓	✓	✓	Partial	✓	OA	✗
DynamiTe [Le et al. 2020]	✓	✓	✓	Partial	✓	OA	✗
RevTerm [Chatterjee et al. 2021]	✗	✓	✓	✗	✓	OA	✗
AProVE [Hensel et al. 2022]	✗	✓	✓	✗	✓	OA	✗
Ultimate [Heizmann et al. 2014]	✓	✗	✓	✗	✓	OA	✗
Pulse$^{\infty}$	✗	✓	✓	✓	✓	UA	✓

Table 1. Positioning Pulse$^{\infty}$ amongst (non)-termination analysis tools in the literature

UNTer$^{\mathrm{SL}}$ are identical to their UNTer counterparts, except that the UNTer states are replaced with corresponding UNTer$^{\mathrm{SL}}$ states to account for heaps. We show that the BUA, FUA and divergent proof system of UNTer$^{\mathrm{SL}}$ are sound, with the full proof given in §G.

Theorem 12 (UNTer$^{\mathrm{SL}}$ soundness)**.** *The BUA, FUA and divergent proof system of UNTer^{SL} are sound.*

7 Evaluation

To demonstrate the feasibility of UNTer$^{\mathrm{SL}}$ for detecting divergence bugs, we have developed Pulse$^{\infty}$ as an extension of the existing Pulse program analyser (underpinned by the ISL [Raad et al. 2020] theory and compatible with both FUA and BUA reasoning). The most fundamental extensions to Pulse are: 1) addition of the divergent triple; 2) symbolic execution steps corresponding to proof rules for divergence; and 3) a stopping condition corresponding to fixpoints (which is unusual for under-approximation) and the associated "test oracle" for recognising divergence. In addition to these fundamental changes, we needed to make some detailed but conceptually minor alterations to the treatment of Booleans in Pulse: it had optimisations which were sound for proving violation of safety properties, but which interfered with our oracle for divergence.

7.1 Pulse$^{\infty}$ in Context

Table 1 places Pulse$^{\infty}$ in the context of other termination and non-termination tools within the last two decades, where **Non-det.** denotes whether the tool supports non-deterministic programming constructs such as `rand()`, **Auto.** denotes whether it is fully automated, and **UA/OA** denotes whether it performs under-approximate (UA) or over-approximate (OA) analysis. To our knowledge, Pulse$^{\infty}$ is the first tool for non-termination analysis with full support for heap reasoning. Tools such as Mutant [Berdine et al. 2006] and Caber [Brotherston and Gorogiannis 2014] are also capable of heap reasoning using separation logic, however these tools were specifically developed to perform over-approximate (OA) termination analysis, and do not support non-termination analysis. Other tools such as TNT [Gupta et al. 2008] and HipTNT [Le et al. 2014, 2015] have limited support for heap reasoning which does not include array or string capabilities.

Most importantly, as we discuss below, to our knowledge Pulse$^{\infty}$ is *the first tool* that can automatically prove non-termination on *large code bases and libraries* such as OpenSSL. As we discuss in §8, several existing tools [Brockschmidt et al. 2013; Chatterjee et al. 2021; Hensel et al. 2022; Kroening et al. 2010; Larraz et al. 2014; Le et al. 2020; Velroyen and Rümmer 2008], can only handle *integer C programs* and thus do not support programs *with function calls*. In other words, unlike Pulse$^\infty$, these tools cannot be run on large C codebases or libraries such as OpenSSL. In particular, Pulse$^\infty$ inherits all the *incremental* capabilities of Pulse (as documented on https://fbinfer.com/docs/next/steps-for-ci#differential-workflow). As a point of reference, analysing OpenSSL on 30 cores with Pulse takes 1m26s the first time. After modifying a file in the project, re-analysing OpenSSL again in incremental mode takes only 6s.

On this point we consulted with authors of the tools referenced in the table. None responded in the affirmative that they could run on large projects. Several tools [Le et al. 2014, 2015] stipulate the existence of a `main` procedure (thus excluding libraries), while others [Heizmann et al. 2014] require that the input to the tool be *a single C file*, and thus one must put the target program and all its dependencies into a single file (thus precluding large code bases and libraries). One theoretical possibility is to automatically construct a fake `main` procedure which calls methods in a representative fashion, but this would get us into a problem similar to *harness generation* in fuzzing, itself a challenging problem and a source of false positives. Our impression overall is that, far from being a simple matter, each of the other tools is one or several research projects away from automatic application at scale.

We evaluate Pulse$^\infty$ in two ways. First, in §7.2 we compare its ability to detect non-termination bugs on small examples and compare it against the state-of-the-art tools. Second, in §7.3 we run Pulse$^\infty$ on several large projects and libraries, comprising over 1.3 million lines of code. Note that as discussed above (and detailed later in §8), no existing automated tool for divergence analysis can be applied to large code bases or libraries *out of the box*, and Pulse$^\infty$ is the only such tool. As such, we could not compare the Pulse$^\infty$ performance against the state of the art for analysing libraries. We refer the reader to our project page for more detailed information about Pulse$^\infty$ and our benchmarks for evaluating it [Raad et al. 2024a].

Experimental Setup. We ran all our experiments below on a single server equipped with an AMD EPYC 7543P processor at 3.4 Ghz clock on 30 active cores (make -j 30) and 512GB of RAM.

7.2 Evaluating Pulse$^\infty$ on Small Examples

SV-COMP Benchmarks. To evaluate Pulse$^\infty$ against existing tools for detecting divergence, we focused on the state-of-the-art termination and divergence non-linear arithmetic benchmarks extending the Competition on Software Verification (SV-COMP) initiative. DynamiTe [Le et al. 2020] outperforms all other tools listed in Table 1. Specifically, at the time of publication, Le et al. [2020] demonstrated that DynamiTe outperformed AProVE [Hensel et al. 2022] for non-termination analysis, and AProVE was in turn shown to outperform all other pre-existing tools in Table 1. As such, rather than comparing Pulse$^\infty$ against all tools listed in Table 1, we compare it directly to DynamiTe and its non-linear arithmetic (NLA) benchmark extension distributed with SV-COMP.

We present our comparison result against DynamiTe in Table 2. The tests listed are rather small and contain at most three loops, with the majority of them comprising a single loop. Both DynamiTe and Pulse$^\infty$ analyse these tests within a few seconds, with the time difference being insignificant. As such, in Table 2 we do not compare the time-performance of the two tools, and focus instead on their reported outcomes. As shown, Pulse$^\infty$ performs competitively against DynamiTe, though the results are not directly comparable. Specifically, Pulse$^\infty$ successfully found divergence bugs that were undetected by DynamiTe (e.g. the dijkstraX-both-nt tests), while it missed others found by DynamiTe (e.g. the geoX-both-nt tests). Notably, thanks to its under-approximate nature, Pulse$^\infty$ reported *zero false positives* (FP), in contrast to DynamiTe which suffered several false positives.

Test	T/NT	Pulse$^\infty$	DynamiTe	Test	T/NT	Pulse$^\infty$	DynamiTe
bresenham1-both-nt	NT	FN	✓	freire1-both-nt	NT	✓	FN
cohencu1-both-nt	NT	FN	✓	geo1-both-nt	NT	FN	✓
cohencu2-both-nt	NT	FN	FN	geo2-both-nt	NT	FN	✓
cohencu3-both-nt	NT	FN	FN	geo3-both-nt	NT	FN	✓
cohencu4-both-nt	NT	FN	FN	hard2-both-nt	NT	FN	✓
cohencu5-both-nt	NT	FN	✓	hard-both-nt	NT	✓	✓
dijkstra1-both-nt-2	NT	FN	FN	hard-both-t	T	?	FP
dijkstra1-both-nt	NT	✓	FN	knuth-both-nt	NT	FN	FN
dijkstra2-both-nt	NT	✓	FN	knuth-nosqrt-both-nt	NT	FN	✓
dijkstra3-both-nt	NT	✓	FN	lcm1-both-t	T	?	FP
dijkstra4-both-nt	NT	✓	FN	lcm1-both-nt	NT	✓	✓
dijkstra5-both-nt	NT	✓	FN	lcm2-both-nt	NT	✓	✓
dijkstra6-both-nt	NT	✓	FN	mannadiv-both-nt	NT	✓	FN
divbin1-both-nt	NT	FN	FN	prod4br-both-nt	NT	FN	FN
egcd2-both-nt	NT	✓	✓	prodbin-both-nt	NT	FN	FN
egcd3-both-t	T	?	FP	ps2-both-nt	NT	FN	✓
egcd3-both-nt	NT	✓	✓	ps3-both-nt	NT	FN	FN
egcd-both-nt	NT	✓	✓	ps4-both-nt	NT	FN	✓
fermat1-both-t	T	?	FP	ps5-both-nt	NT	FN	FN
fermat1-both-nt	NT	FN	FN	ps6-both-nt	NT	FN	FN
fermat2-both-nt	NT	✓	✓	sqrt1-both-nt	NT	FN	✓
fermat3-both-nt	NT	FN	✓	sqrt2-both-nt	NT	FN	FN
Total		**Pulse$^\infty$:** 15 ✓, 25 FN, 0 FP			**DynamiTe:** 19 ✓, 21 FN, 4 FP		

Table 2. Comparing Pulse$^\infty$ against DynamiTe on SV-COMP non-linear arithmetic (NLA) benchmarks for termination (T) and non-termination (NT). For NT cases, ✓ denotes a true positive (i.e. the tool correctly detected non-termination) and FN a false negative (i.e. the tool failed to detect non-termination. For T cases, ? under Pulse$^\infty$ denotes an unknown result (Pulse$^\infty$ proves non-termination and not termination), and FP under DynamiTe denotes a false positive (i.e. DynamiTe incorrectly reported that the program terminates).

From this evaluation we cannot conclude that Pulse$^\infty$ has superior precision on small examples compared to the state of the art, but neither can we conclude that it is inferior. Note that, because Pulse$^\infty$'s theory is sound and complete, and it directly represents the fundamental recurrence set idea of Gupta et al. [2008] (see 8), we *could* in principle import any algorithmic techniques that appear in other papers, for the purpose of establishing divergence triples. So the precision here is an implementation rather than a fundamental matter, and the purpose of the evaluation on the small benchmarks was just to confirm that the Pulse$^\infty$ is not too far off the state of the art, and having done so this sets us up for the more significant evaluation on larger projects.

7.3 Running Pulse$^\infty$ on Large Projects and Libraries

We evaluated Pulse$^\infty$ on a number of large open source projects including OpenSSL, libxml2, CryptoPP and libxpm. We present the result of our analysis in Table 3. As shown, each of the libraries analysed comprises thousands of lines of code (LOC) and were each analysed within minutes. In total, we identified and reported eleven previously unknown divergence bugs in OpenSSL, libxml2, CryptoPP and libxpm. Several of these bugs have been acknowledged with fixes waiting to be merged, while others are under discussion in the bug tracking system. Some of the issues we reported in OpenSSL are instances of *latent* (non-manifest) errors according to the classification by Le et al. [2022]: they are only reachable when the culprit function is called with

Library	Language	#LOC analysed	Time	# Bugs reported
OpenSSL	C	804 K	1m, 26s	4
libxml2	C	300 K	1m, 3s	4
CryptoPP	C++	51 K	2m, 40s	2
libxpm	C	11 K	2s	1
libpng	C	96 K	6s	0
zlib	C	41 K	1m, 7s	0
ngiflib	C	1.7 K	1s	0
Total		**1.3 M**	**14m, 5s**	**11**

Table 3. Evaluating Pulse$^{\infty}$ on large projects (1.3 million lines of code analysed and 11 new bugs found)

specific parameter values. Until these latent conditions are removed, it remains up to the caller to enforce well-behaved input to the problematic callee functions.

In the cases of CryptoPP and libxpm we have submitted pull requests with patches to be merged in due course. We also reported our findings in libxml2 on its project bug tracking system, where our discussion with project maintainers suggests that while the vulnerable functions are active, the specific divergent conditional branches in them are never executed and can be safely removed.

We present a bug we found in CryptoPP (a popular cryptographic toolkit in C++) in Listing 2. The procedure shown attempts to allocate memory in a loop (lines 4–9). However, as there is no guarantee that `malloc` will succeed and return a non-null value, the loop may not terminate. Our proposed fix is shown at lines 2, 5 and 7 (prefixed with '+'), where we record the number of unsuccessful allocations in `cnt` and throw an exception after 10 attempts.

```
1    void* AlignedAllocate(size_t size) {
2    +  unsigned int cnt = 0;
3       byte *p;
4       while ((p = (byte *)malloc(size+16)) == NULLPTR) {
5    +     if (cnt >= 10) { throw std::bad_alloc(); }
6          CallNewHandler();
7    +     cnt++;
8       }
9       CRYPTOPP_ASSERT(IsAlignedOn(p, 16));
10      return p;  }
```

Listing 2. A divergence bug found by Pulse$^{\infty}$ in CryptoPP, with our proposed fix given by adding the '+'-prefixed lines.

8 Related Work

Termination and Non-Termination Tools. There are many individual reports of divergence bugs which many readers will no doubt relate to. Notably, a recent empirical study on OSS projects found 445 non-termination bugs from 3,142 GitHub commits [Shi et al. 2022].

There has been significant work on automated methods for proving termination; see the survey by Cook et al. [2011]. 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 [2005] argues, 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. [2008] uses proof to find divergence bugs. They use a transition system with initial/final states and a transition relation, and they identify the notion 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 $\vdash_{\mathrm{B}} [R]\ \mathsf{C}\ [ok\colon R]$. One might argue that the relation between the UNTer proof system for $\vdash_{\mathrm{B}} [p]\ \mathsf{C}\ [ok\colon q]$ and the model of Gupta et al. [2008] is analogous to the relation between Hoare's logic and Floyd's proof method [Apt and

Olderog 2019]: using under-approximate triples enables compositional reasoning. There are many detailed differences beyond these. They first run a concolic executor to gather assertions at program points, especially loop entry, but then employ an arithmetic encoding to derive reachability facts for loop bodies, and they treat the heap concretely (as the encoding is difficult otherwise). By contrast, we reason about reachability both of the loop stems and bodies within UNTer^{SL}, and we harness separation logic (SL) to reason abstractly about heaps (SL-based analyses were not available at the time of the work by Gupta et al. [2008]).

Our prototype, Pulse$^{\infty}$, inherits the strengths and weaknesses of Pulse. Specifically, it is easy to run Pulse on program snippets, to scale it to large programs, and to incorporate it in a CI-based deployment on pull requests. On the other hand, Pulse has a weak treatment of arithmetic, meaning that some complex examples (as in the work of Gupta et al. [2008]) may not be provable. The strengths and weaknesses of Gupta et al. [2008] are the converse. We do not believe the weaknesses of either are inevitable; e.g. by adding a stronger arithmetic solver to Pulse$^{\infty}$ it would be possible to prove the more complex examples; the question is the effect this would have on performance. Upon contacting Gupta et al. [2008], we were informed that *their tool is no longer available*; as such, we were unable to compare Pulse$^{\infty}$ against it.

After Gupta et al. [2008], there have been many further papers on automatically proving/checking termination/non-termination. Cook et al. [2015] and Chen et al. [2014] introduce novel ideas on the use of over-approximation, going beyond the under-approximate logics here.

Many existing tools [Brockschmidt et al. 2013; Chatterjee et al. 2021; Kroening et al. 2010; Larraz et al. 2014; Velroyen and Rümmer 2008] focus on the syntax of termination problems defined by the Termination Competition and can *only* handle *integer C programs*, i.e. programs 1) with only integer datatypes; and 2) *without function calls*. As a result, such tools *cannot* run on programs that do not conform to this syntax, unless they are first pre-processed into integer C programs. In other words, unlike Pulse$^{\infty}$, these tools cannot run on existing C codebases or libraries such as OpenSSL.

The T2 tool by Cook et al. [2013] can be used to prove termination (and not divergence). However, T2 does not support heaps, and thus (unlike Pulse$^{\infty}$) cannot handle examples where termination is due to e.g. pointer arithmetic. Moreover, as our direct communications with the authors have revealed, T2 requires a C front-end, and the front-end the authors used bit-rotted' a while ago. The AProVE tool by Hensel et al. [2022] uses T2 as one of its back-ends for analysing termination; however, AProVE only supports integer C programs (as discussed above). Furthermore, upon contacting the authors about running AProVE on large libraries, we were informed that their techniques “are quite precise and do not have sufficient abstraction methods for handling very large programs within reasonable time”. Most significantly, however, AProVE only supports programs that contain a `main()` method, and thus *cannot be used to analyse libraries* such as OpenSSL. This is in contrast to Pulse$^{\infty}$, where we successfully analysed hundreds of thousands of lines of code within minutes, and could effortlessly analyse libraries such as OpenSSL and libxml2. Heizmann et al. [2014] have extended the Ultimate framework to detect divergence using Büchi automata. However, as confirmed by the authors, a key technical limitation of their tool is that the input must be *a single C file* (and thus the program being analysed and all its dependencies must be included in one file), thus precluding its application to large projects and libraries comprising numerous modules and dependencies spread across several files.

Brotherston and Gorogiannis [2014] present Caber for proving termination (not divergence). While Caber supports heaps, it has only been applied to a handful of small programs, and not large code bases or libraries. Le et al. [2014, 2015] present the HipTNT and HipTNT+ tools for proving termination and non-termination. However, as we were informed in the course of our direct communication with the authors, these tools can only handle small programs such as those in the SV-COMP benchmarks. Moreover, these tools require the user to supply certain annotations and are thus not fully automated. Le et al. [2020] later adapted these tools to develop DynamiTe, a dynamic termination analyser for non-linear program. However, DynamiTe can only handle integer programs (as discussed above.)

The idea of detecting divergence using proof is appealing and intuitively not too complicated. Although our work is but a step on the way, it is reasonable to hope that divergence proof techniques may mature to a degree where they can be routinely deployed in engineering practice.

Under-Approximation and Incorrectness. This paper follows a line of work on under-approximate reasoning following incorrectness logic [O'Hearn 2019], but is part of a more extensive history. O'Hearn [2019] used FUA triples to reason about incorrectness and to avoid false positives. FUA triples were studied previously by de Vries and Koutavas [2011], but they did not make the connection to incorrectness or absence of false positives. Further, as O'Hearn [2019] remarked, FUA triples could be expressed in dynamic logic [Harel 1979] with a backwards diamond modality (or forwards with a transition reversal operator). Moreover, they are similar to the must^{-} transitions used by e.g. [Ball et al. 2005]. As such, the FUA triple is not itself novel, but its significance has been uncovered gradually.

Here we study both BUA and FUA triples. BUA triples were mentioned by de Vries and Koutavas [2011] under the name "total Hoare triple", but were not studied by them. These triples can also be expressed immediately in dynamic logic, without resorting to backwards modalities or reversal, and they are similar to the must^{+} transitions used of [Ball et al. 2005]. More recently BUA triples were suggested by Derek Dreyer and Ralf Jung just before IL was formulated, but they remained unexplored. (This was during a discussion with Peter O'Hearn and Jules Villard at POPL 2019 in Lisbon, and thus BUA triples are also informally referred to as 'Lisbon' triples in the literature.) BUA triples are also studied by Möller et al. [2021], but only their metatheory and not their applications. Zilberstein et al. [2023] developed Outcome Logic (OL) where they make meta-theoretic remarks on how BUA could serve as a foundation for incorrectness, but they do not demonstrate the practical advantages or disadvantages of such reasoning. Specifically, while they discuss the merits of BUA for identifying manifest errors, they do not demonstrate the practical impact of this e.g. in a scalable analysis tool. Nor do they explore the advantage of BUA for non-termination analysis. Zilberstein et al. [2024] later extend OL with separation logic. Ascari et al. [2024] also study BUA triples in sufficient incorrectness logic (SIL). As with FUA, the notion of BUA is not itself novel, but its significance is emerging gradually. This paper adds two new insights about BUA.

The first is that abducing preconditions is, in a sense, forced on us if we are to do forward reasoning with BUA. This is because BUA triples are not closed under postconditions: given a precondition p and a program C, there need not exist a corresponding postcondition making the triple valid. This is the case for any p which has a state on which C always diverges. As a result, there is no analogue for BUA of Dijkstra's strongest postcondition predicate transformer (where this transformer works for FUA). This would, at first glance, make BUA appear problematic for forward reasoning: forward reasoning in abstract interpretation uses over-approximations of Dijkstra's transformer, reasoning with FUA can use under-approximation of it, and this abstraction-of-post tactic is not available for BUA. It might be possible to automate backward reasoning instead (find preconditions given a program and a post), but there is another possibility: abduction. Given a precondition and a program, we can try to abduce an addition $?A$ to the precondition to guarantee the existence of a postcondition. For example, for $\vdash_{\mathrm{B}} \left[\mathit{true} \land ?A\right]$ $\mathtt{if(even(}x\mathtt{))\ diverge\ else\ }x\mathtt{:=}x\mathtt{+1}$ $\left[\mathit{ok}\colon ?B\right]$, we can abduce $?A = \mathtt{odd}(x)$ to give us a precondition to establish the postcondition $?B\ = \mathtt{even}(x)$.

Abduction is used to reason forwards with BUA triples to skirt the absence of general postconditions. This is not unlike the case in classic separation logic (SL), where the absence of general postconditions for the fault-avoiding interpretation of SL triples is circumvented by abducing safe preconditions [Calcagno et al. 2011]. We emphasise that the point we are making here is stronger than the mere *compatibility* of BUA with abduction (which has been recognised by Zilberstein et al. [2024]): we cannot reason forward from an arbitrary preconditon, without changing it. For comparison, consider the situation with FUA. In FUA we can do abduction, it is compatible with it, but since FUA is closed under postconditions (there is a post for any given precondition) we could in principle reason forwards if we so desired, without using abduction to shrink the precondition.

The second BUA insight in our work is that (unlike in other works [Ascari et al. 2024; de Vries and Koutavas 2011; Möller et al. 2021; Zilberstein et al. 2023, 2024]) we demonstrate its suitability for reasoning about non-termination, where pure FUA is unsound for non-termination proof rules. This goes together with the ability of BUA triples to *weaken a postcondition*, which opens up the possibility of iterating to a fixed-point as suggested in the main proof rule for diverging loops. Thus, BUA tools can share some of the iteration strategies with their over-approximate cousins, and in contrast to FUA-only tools. A crucial difference is that the fixpoint then implies divergence.

It is important to note that there are disadvantages to BUA-only approaches. For example, BUA does not support shrinking a postcondition, which would block the application of partial concretisation as in Klee, DART and similar tools (see the work of [O'Hearn 2019]). Indeed, must$^-$ transitions, a relative of FUA triples, have been used to formalise the reasoning in such tools [Godefroid et al. 2010]. Note that there is a more basic under-approximate triple, let us call it the existential (EUA) triple written as $\vdash_{\mathsf{E}}$ $[P]$ C $[\epsilon : Q]$, stating that *some* state in P reaches *some* state in Q by executing C. EUA is sufficient for proving incorrectness and avoiding false positives, so why is it not the basis of a program logic? The problem is that EUA is not closed under sequential composition (the Seq rule fails), which makes reasoning about paths challenging. Both BUA and FUA triples are closed under sequential composition, and this is (we presume) why they have received more attention. But, the composition of a BUA followed by a FUA triple establishes an EUA triple; BUA and FUA can be used together [Ball et al. 2005].

Due to the reasons discussed above, it is better for a tool, or a portion of a tool, to be compatible with both BUA and FUA, rather than one or the other. As observed here, the Pulse framework is compatible with both. Extension to non-termination or concretisation should take into account considerations as above. While the BUA and FUA metatheory seems mostly settled, we do not claim that the above remarks fully account for their strengths, weaknesses or overall significance for reasoning: we are still learning about them.

Data Availability Statement

The proofs of all stated theorems in the paper are given in the appendix. Our prototype tool Pulse$^\infty$ is open-source and available as an artifact online [Raad et al. 2024b].

Acknowledgments

We thank the anonymous OOPSLA 2024 reviewers for their constructive feedback, and thank Viktor Vafeiadis and Jules Villard for many fruitful discussions. Azalea Raad is supported by the UKRI Future Leaders Fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1 and by VeTSS.

References

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

Flavio Ascari, Roberto Bruni, Roberta Gori, and Francesco Logozzo. 2024. Sufficient Incorrectness Logic: SIL and Separation SIL. arXiv:2310.18156 [cs.LO] https://arxiv.org/abs/2310.18156

Thomas Ball, Orna Kupferman, and Greta Yorsh. 2005. Abstraction for Falsification. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (Lecture Notes in Computer Science)*, Kousha Etessami and Sriram K. Rajamani (Eds.), Vol. 3576. Springer, 67–81. https://doi.org/10.1007/11513988_8

Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. 2007. Variance Analyses from Invariance Analyses. *SIGPLAN Not.* 42, 1 (jan 2007), 211–224. https://doi.org/10.1145/1190215.1190249

Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2006. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276514

Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better termination proving through cooperation. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25.* Springer, 413–429.

James Brotherston and Nikos Gorogiannis. 2014. Cyclic Abduction of Inductively Defined Safety and Termination Preconditions. In *Static Analysis*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer International Publishing, Cham, 68–84.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. http://doi.acm.org/10.1145/2049697.2049700

Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotnỳ, and Đorđe Žikelić. 2021. Proving non-termination by program reversal. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 1033–1048.

Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. 2008. Ranking Abstractions. In *Programming Languages and Systems*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 148–162.

Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 156–171. https://doi.org/10.1007/978-3-642-54862-8_11

Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2015. Embracing Overapproximation for Proving Nontermination. *Tiny Trans. Comput. Sci.* 3 (2015). http://tinytocs.org/vol3/papers/TinyToCS_3_cook.pdf

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006a. Termination Proofs for Systems Code. *SIGPLAN Not.* 41, 6 (jun 2006), 415–426. https://doi.org/10.1145/1133255.1134029

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006b. Terminator: Beyond Safety. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 415–418.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving program termination. *Commun. ACM* 54, 5 (2011), 88–98. https://doi.org/10.1145/1941487.1941509

Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. lexicographic termination proving. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19.* Springer, 47–61.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular Termination Verification for Non-blocking Concurrency. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–201.

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings.* 155–171. https://doi.org/10.1007/978-3-642-24690-6_12

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-Grained Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 16 (nov 2021), 134 pages. https://doi.org/10.1145/3477082

Patrice Godefroid. 2005. The soundness of bugs is what matters (position statement). https://www.cs.umd.edu/~pugh/BugWorkshop05/papers/11-godefroid.pdf

Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 43–56. https://doi.org/10.1145/1706299.1706307

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* *POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 147–158. https://doi.org/10.1145/1328438.1328459

David Harel. 1979. *First-Order Dynamic Logic.* Springer-Verlag, Berlin, Heidelberg.

William R Harris, Akash Lal, Aditya V Nori, and Sriram K Rajamani. 2010. Alternation for termination. In *Static Analysis: 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings 17.* Springer, 304–319.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 797–813.

Jera Hensel, Constantin Mensendiek, and Jürgen Giesl. 2022. AProVE: Non-Termination Witnesses for C Programs: (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 403–407.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL)*. Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/360204.375719

Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M Wintersteiger. 2010. Termination analysis with compositional transition invariants. In *International Conference on Computer Aided Verification.* Springer, 89–103.

Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2014. Proving non-termination using Max-SMT. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26.* Springer, 779–796.

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (apr 2022), 27 pages. https://doi.org/10.1145/3527325

Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 189 (nov 2020), 30 pages. https://doi.org/10.1145/3428257

Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. 2014. A resource-based logic for termination and non-termination proofs. In *Formal Methods and Software Engineering: 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings 16.* Springer, 267–283.

Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and non-termination specification inference. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 489–498. https://doi.org/10.1145/2737924.2737993

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects under Fair Scheduling. *SIGPLAN Not.* 51, 1 (jan 2016), 385–399. https://doi.org/10.1145/2914770.2837635

Bernhard Möller, Peter W. O'Hearn, and Tony Hoare. 2021. On Algebra of Program Correctness and Incorrectness. In *Relational and Algebraic Methods in Computer Science - 19th International Conference, RAMiCS 2021, Marseille, France, November 2-5, 2021, Proceedings (Lecture Notes in Computer Science)*, Uli Fahrenberg, Mai Gehrke, Luigi Santocanale, and Michael Winter (Eds.), Vol. 13027. Springer, 325–343. https://doi.org/10.1007/978-3-030-88701-8_20

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. http://doi.acm.org/10.1145/3371078

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 225–252.

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent Incorrectness Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 34 (jan 2022), 29 pages. https://doi.org/10.1145/3498695

Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. 2023. A General Approach to Under-Approximate Reasoning About Concurrent Programs. In *34th International Conference on Concurrency Theory (CONCUR 2023) (Leibniz International Proceedings in Informatics (LIPIcs))*, Guillermo A. Pérez and Jean-François Raskin (Eds.), Vol. 279. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:17. https://doi.org/10.4230/LIPIcs.CONCUR.2023.25

Azalea Raad, Julien Vanegue, and Peter O'Hearn. 2024a. Extended Version. https://www.soundandcomplete.org/papers/OOPSLA2024/Unter/

Azalea Raad, Julien Vanegue, and Peter O'Hearn. 2024b. The Pulse$^\infty$ prototype tool. https://doi.org/10.5281/zenodo.12637589

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. 2022. Large-scale analysis of non-termination bugs in real-world OSS projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 256–268. https://doi.org/10.1145/3540250.3549129

Helga Velroyen and Philipp Rümmer. 2008. Non-termination checking for imperative programs. In *International Conference on Tests and Proofs*. Springer, 154–170.

Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (apr 2023), 29 pages. https://doi.org/10.1145/3586045

Noam Zilberstein, Angelina Saliling, and Alexandra Silva. 2024. Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 104 (apr 2024), 29 pages. https://doi.org/10.1145/3649821

A Divergence Vulnerabilities

Divergence bugs are widespread across a number of programming languages. We present several examples taken from the Common Vulnerabilities and Exposures (CVE) database and categorise them along common cases of vulnerabilities – see Fig. 9 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 behaviours where non-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.

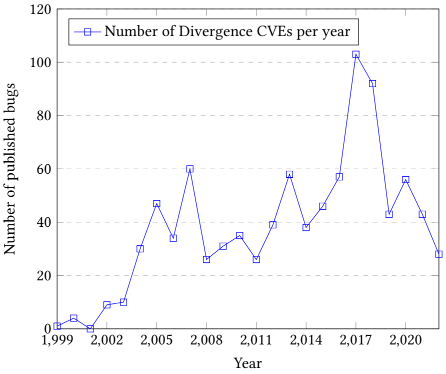

Fig. 9. Vulnerability trend for divergence bugs

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 §H.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 §H.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 §H.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 §H.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-bounds element is read (see §H.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 incremented/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 §H.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 §H.7). This example illustrates that even security code can be vulnerable to divergence bugs!

B Derived Rules

IfTrue Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\dagger}[p \land B]\ \mathsf{assume}(B)\ [ok\colon p \land B]}\,(\textsc{Assume}) \qquad \dfrac{}{\vdash_{\dagger}[p \land B]\ \mathsf{C}_1\ [ok\colon q]}\,(\text{given})}{\vdash_{\dagger}[p \land B]\ \mathsf{assume}(B);\mathsf{C}_1\ [\epsilon : q]}\,(\textsc{Seq})}{\vdash_{\dagger}[p \land B]\ (\mathsf{assume}(B);\mathsf{C}_1) + (\mathsf{assume}(\neg B);\mathsf{C}_2)\ [\epsilon : q]}\,(\textsc{Choice})}{\vdash_{\dagger}[p \land B]\ \mathsf{if}\ (B)\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2\ [\epsilon : q]}\,(\text{If encoding})$$

IfFalse Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\dagger}[p \land \neg B]\ \mathsf{assume}(\neg B)\ [ok\colon p \land \neg B]}\,(\textsc{Assume}) \qquad \dfrac{}{\vdash_{\dagger}[p \land \neg B]\ \mathsf{C}_2\ [ok\colon q]}\,(\text{given})}{\vdash_{\dagger}[p \land \neg B]\ \mathsf{assume}(\neg B);\mathsf{C}_2\ [\epsilon : q]}\,(\textsc{Seq})}{\vdash_{\dagger}[p \land \neg B]\ (\mathsf{assume}(B);\mathsf{C}_1) + (\mathsf{assume}(\neg B);\mathsf{C}_2)\ [\epsilon : q]}\,(\textsc{Choice})}{\vdash_{\dagger}[p \land \neg B]\ \mathsf{if}\ (B)\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2\ [\epsilon : q]}\,(\text{If encoding})$$

ConsEq Derivation (BUA case)

$$\dfrac{\dfrac{\dfrac{}{p \Leftrightarrow p'}\,(\text{given})}{p \subseteq p'} \qquad \dfrac{}{\vdash_{\mathsf{B}}\ [p']\ \mathsf{C}\ [\epsilon : q']}\,(\text{given}) \qquad \dfrac{\dfrac{}{q \Leftrightarrow q'}\,(\text{given})}{q' \subseteq q}}{\vdash_{\mathsf{B}}\ [p]\ \mathsf{C}\ [\epsilon : q]}\,(\textsc{ConsF})$$

ConsEq Derivation (FUA case)

$$\dfrac{\dfrac{\dfrac{}{p \Leftrightarrow p'}\,(\text{given})}{p' \subseteq p} \qquad \dfrac{}{\vdash_{\mathsf{F}}\ [p']\ \mathsf{C}\ [\epsilon : q']}\,(\text{given}) \qquad \dfrac{\dfrac{}{q \Leftrightarrow q'}\,(\text{given})}{q \subseteq q'}}{\vdash_{\mathsf{F}}\ [p]\ \mathsf{C}\ [\epsilon : q]}\,(\textsc{ConsB})$$

WhileFalse Derivation

$$\dfrac{\dfrac{\dfrac{}{\vdash_{\dagger}[p \land \neg B]\ (\mathsf{assume}(B);\mathsf{C})^{\star}\ [ok\colon p \land \neg B]}\,(\textsc{Loop0}) \qquad \dfrac{}{\vdash_{\dagger}[p \land \neg B]\ \mathsf{assume}(\neg B)\ [ok\colon p \land \neg B]}\,(\textsc{Assume})}{\vdash_{\dagger}[p \land \neg B]\ (\mathsf{assume}(B);\mathsf{C})^{\star};\mathsf{assume}(\neg B)\ [ok\colon p \land \neg B]}\,(\textsc{Seq})}{\vdash_{\dagger}[p \land \neg B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [ok\colon p \land \neg B]}\,(\text{while encoding})$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{(1) \quad (2)}{\vdash_{\dagger}[p(0) \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star};\mathsf{assume}(B);\mathsf{C}\ [ok\colon q \land \neg B]}\,(\textsc{Seq})}{\vdash_{\dagger}[p(0) \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star}\ [ok\colon q \land \neg B]}\,(\textsc{Loop}) \qquad \dfrac{}{\vdash_{\dagger}[q \land \neg B]\ \mathsf{assume}(\neg B)\ [ok\colon q \land \neg B]}\,(\textsc{Assume})}{\vdash_{\dagger}[p(0) \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star};\mathsf{assume}(\neg B)\ [ok\colon q \land \neg B]}\,(\textsc{Seq})}{\vdash_{\dagger}[p(0) \land B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [ok\colon q \land \neg B]}\,(\text{while encoding})$$

with

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall n < k.\ \vdash_{\dagger} [r(n)]\ \mathsf{assume}(B)\ [ok\colon r(n)]}\ \text{Assume} \qquad \dfrac{}{\forall n < k.\ \vdash_{\dagger} [r(n)]\ \mathsf{C}\ [ok\colon r(n{+}1)]}\ (\text{given})}{\forall n < k.\ \vdash_{\dagger} [r(n)]\ \mathsf{assume}(B);\mathsf{C}\ [ok\colon r(n{+}1)]}\ (\text{Seq})}{\vdash_{\dagger} [r(0)]\ (\mathsf{assume}(B);\mathsf{C})^{\star}\ [ok\colon r(k)]}\ (\text{Loop-Subvar})}{\vdash_{\dagger} [p(0) \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star}\ [ok\colon p(k) \land B]}\ (\text{definition of } r)$$

(1)

and

$$\dfrac{\dfrac{}{\vdash_{\dagger} [p(k) \land B]\ \mathsf{assume}(B)\ [ok\colon p(k) \land B]}\ (\text{Assume}) \qquad \dfrac{}{\vdash_{\dagger} [p(k) \land B]\ \mathsf{C}\ [ok\colon q \land \neg B]}\ (\text{given})}{\vdash_{\dagger} [p(k) \land B]\ \mathsf{assume}(B);\mathsf{C}\ [ok\colon q \land \neg B]}\ (\text{Seq})$$

(2)

Div-LoopNest Derivation

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

$$\dfrac{\dfrac{\dfrac{}{[p]\ \mathsf{C}\ [\infty]}\ (\text{given})}{[p]\ \mathsf{C};\mathsf{C}^{\star}\ [\infty]}\ (\text{Div-Seq1})}{[p]\ \mathsf{C}^{\star}\ [\infty]}\ (\text{Div-LoopUnfold})$$

Div-While Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\mathsf{B}} [p \land B]\ \mathsf{assume}(B)\ [ok\colon p \land B]}\ (\text{Assume}) \qquad \dfrac{}{\vdash_{\mathsf{B}} [p \land B]\ \mathsf{C}\ [ok\colon q \land B]}\ (\text{given})}{\vdash_{\mathsf{B}} [p \land B]\ \mathsf{assume}(B);\mathsf{C}\ [ok\colon q \land B]}\ (\text{Seq}) \qquad \dfrac{\dfrac{}{q \subseteq p}\ (\text{given})}{q \land B \subseteq p \land B}}{[p \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star}\ [\infty]}\ (\text{Div-Loop})}{\dfrac{[p \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star};\mathsf{assume}(\neg B)\ [\infty]}{[p \land B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\infty]}\ (\text{while encoding})}\ (\text{Div-Seq1})$$

Div-WhileNest Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\mathsf{B}} [p \land B]\ \mathsf{assume}(B)\ [ok\colon p \land B]}\ (\text{Assume}) \qquad \dfrac{}{[p \land B]\ \mathsf{C}\ [\infty]}\ (\text{given})}{[p \land B]\ \mathsf{assume}(B);\mathsf{C}\ [\infty]}\ (\text{Div-Seq2})}{[p \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star}\ [\infty]}\ (\text{Div-LoopNest})}{\dfrac{[p \land B]\ (\mathsf{assume}(B);\mathsf{C})^{\star};\mathsf{assume}(\neg B)\ [\infty]}{[p \land B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\infty]}\ (\text{while encoding})}\ (\text{Div-Seq1})$$

Div-WhileSubvar Derivation

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

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(\textsc{Assume})}
\UnaryInfC{$\forall n \in \mathbb{N}.\ \vdash_{\mathrm{B}} [q(n)]\ \mathsf{assume}(B)\ [ok\colon q(n)]$}
\AxiomC{}
\RightLabel{(given)}
\UnaryInfC{$\forall n \in \mathbb{N}.\ \vdash_{\mathrm{B}} [q(n)]\ \mathsf{C}\ [ok\colon q(n{+}1)]$}
\RightLabel{(\textsc{Seq})}
\BinaryInfC{$\forall n \in \mathbb{N}.\ \vdash_{\mathrm{B}} [q(n)]\ (\mathsf{assume}(B); \mathsf{C})\ [ok\colon q(n{+}1)]$}
\RightLabel{(\textsc{Div-Subvar})}
\UnaryInfC{$[q(0)]\ (\mathsf{assume}(B); \mathsf{C})^{\star}\ [\infty]$}
\doubleLine
\RightLabel{(definition of $q(0)$)}
\UnaryInfC{$[p(0) \land B]\ (\mathsf{assume}(B); \mathsf{C})^{\star}\ [\infty]$}
\RightLabel{(\textsc{Div-Seq1})}
\UnaryInfC{$[p(0) \land B]\ (\mathsf{assume}(B); \mathsf{C})^{\star}; \mathsf{assume}(\neg B)\ [\infty]$}
\doubleLine
\RightLabel{(while encoding)}
\UnaryInfC{$[p(0) \land B]\ \mathsf{while}\ (B)\ \mathsf{C}\ [\infty]$}
\end{prooftree}
$$

C UNTer Soundness

Proposition 1. *For all* $r, s, \mathrm{C}, n, s', \epsilon$, *if* $s \in r$, $\mathsf{fv}(r) \cap \mathsf{mod}(\mathrm{C}) = \emptyset$ *and* $\mathrm{C}, s \xrightarrow{n} -, s', \epsilon$, *then* $s' \in r$.

Proposition 2. *For all* $\mathbb{C}_1, \mathbb{C}_2, s_1, s_2, s_3$, *if* $\mathbb{C}_1, s_1 \rightsquigarrow^{+}_{\text{er}} \mathsf{skip}, s_2$, *and* $\mathbb{C}_2, s_2 \rightsquigarrow^{+}_{\text{er}} \mathsf{skip}, s_3$, *then* $\mathbb{C}_1; \mathbb{C}_2, s_1 \rightsquigarrow^{+}_{\text{er}} \mathsf{skip}, s_3$.

Proposition 3. *For all* $n, \mathrm{C}, s, s', \epsilon$, *if* $\mathrm{C}^{\star}; \mathrm{C}, s \xrightarrow{n} -, s', \epsilon$ *then there exists* m *such that* $\mathrm{C}; \mathrm{C}^{\star}, s \xrightarrow{m} -, s', \epsilon$.

Lemma 1. *For all* n, s, s', $\mathbb{C}$, $\mathbb{C}'$, *if* $\mathbb{C}, s \xrightarrow{n} \mathbb{C}', s', ok$, *then* $\mathbb{C}' = \mathsf{skip}$.

Proof. By induction on n.

Base case n=0

Pick arbitrary s, s', $\mathbb{C}$, $\mathbb{C}'$ such that $\mathbb{C}, s \xrightarrow{0} \mathbb{C}', s', ok$. From the definition of $\xrightarrow{0}$ we then have $\mathbb{C}'$=skip, as required.

Inductive case n=k+1

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

C.1 Soundness of BUA and FUA Rules

Lemma 2. *For all* s, s', s'', $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}'$, i, j, ϵ, *if* $\mathbb{C}_1, s \xrightarrow{i} -, s'', ok$ *and* $\mathbb{C}_2, s'' \xrightarrow{j} \mathbb{C}', s', \epsilon$, *then there exists* n *such that* $\mathbb{C}_1; \mathbb{C}_2, s \xrightarrow{n} \mathbb{C}', s', \epsilon$.

Proof. Pick arbitrary s, s', s'', $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}'$, $\mathbb{C}''$, i, j, ϵ, such that $\mathbb{C}_1, s \xrightarrow{i} \mathbb{C}'', s'', ok$ and $\mathbb{C}_2, s'' \xrightarrow{j} \mathbb{C}', s', \epsilon$. We proceed by induction on i.

Case $i = 0$

From $\mathbb{C}_1, s \xrightarrow{0} \mathbb{C}'', s'', ok$ we know $\mathbb{C}_1 = \mathbb{C}'' = \mathsf{skip}$ and $s = s''$. As such, since $\mathbb{C}_1 = \mathsf{skip}$ and $s = s''$, from S-SeqSkip we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_2, s'', ok$. Consequently, from $\mathbb{C}_2, s'' \xrightarrow{j} \mathbb{C}', s', \epsilon$ and the definition of $\xrightarrow{j+1}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \xrightarrow{j+1} \mathbb{C}', s', \epsilon$, as required.

Case $i = k$+1

From the definition of $\mathbb{C}_1, s \xrightarrow{i} \mathbb{C}'', s'', ok$ we then know there exists $\mathbb{C}_3, s_3$ such that $\mathbb{C}_1, s \rightarrow \mathbb{C}_3, s_3, ok$ and $\mathbb{C}_3, s_3 \xrightarrow{k} \mathbb{C}'', s'', ok$. As such, from the inductive hypothesis, $\mathbb{C}_3, s_3 \xrightarrow{k} \mathbb{C}'', s'', ok$ and $\mathbb{C}_2, s'' \xrightarrow{j} \mathrm{C}', s', \epsilon$ we know there exists n such that $\mathbb{C}_3; \mathbb{C}_2, s_3 \xrightarrow{n} \mathbb{C}', s', \epsilon$. Moreover, as $\mathbb{C}_1, s \rightarrow \mathbb{C}_3, s_3, ok$, from S-Seq1 we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_3; \mathbb{C}_2, s_3, ok$. Consequently, as $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_3; \mathbb{C}_2, s_3, ok$ and $\mathbb{C}_3; \mathbb{C}_2, s_3 \xrightarrow{n} \mathbb{C}', s', \epsilon$, from the definition of $\xrightarrow{n+1}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \xrightarrow{n+1} \mathbb{C}', s', \epsilon$, as required. □

Lemma 3. *For all* $s, s', s'', \mathbb{C}_1, \mathbb{C}_2, \mathbb{C}', i$, *if* $\mathbb{C}_1, s \xrightarrow{i} \mathbb{C}', s', er$, *and* $\mathbb{C}_2, s' \rightsquigarrow^{+}_{\text{er}} \mathsf{skip}, s''$, *then* $\mathbb{C}_1; \mathbb{C}_2, s \xrightarrow{i} \mathbb{C}'; \mathbb{C}_2, s'', er$.

Proof. Pick arbitrary s, s', s'', $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}'$, i such that $\mathbb{C}_1, s \xrightarrow{i} \mathbb{C}', s', er$ and $\mathbb{C}_2, s' \rightsquigarrow^{+}_{\text{er}} \mathsf{skip}, s''$. We proceed by induction on i.

Case $i = 1$
From $\mathbb{C}_1, s \xrightarrow{1} \mathbb{C}', s', er$ we know there exists s_m such that $\mathbb{C}_1, s \rightarrow \mathbb{C}', s_m, er$ and $\mathbb{C}', s_m \rightsquigarrow^+_{\text{er}} \text{skip}, s'$. As such, from S-Seq1 we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}'; \mathbb{C}_2, s_m, er$. Moreover, from $\mathbb{C}', s_m \rightsquigarrow^+_{\text{er}} \text{skip}, s'$, $\mathbb{C}_2, s' \rightsquigarrow^+_{\text{er}} \text{skip}, s''$ and Prop. 2 we have $\mathbb{C}'; \mathbb{C}_2, s_m \rightsquigarrow^+_{\text{er}} \text{skip}, s''$. Consequently, from $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}'; \mathbb{C}_2, s_m, er$, $\mathbb{C}'; \mathbb{C}_2, s_m \rightsquigarrow^+_{\text{er}} \text{skip}, s''$ and the definition of $\xrightarrow{1}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \xrightarrow{1} \mathbb{C}'; \mathbb{C}_2, s', er$, as required.

Case $i = k{+}1$
From the definition of $\mathbb{C}_1, s \xrightarrow{i} \mathbb{C}', s', er$ we then know there exists $\mathbb{C}_3, s_3$ such that $\mathbb{C}_1, s \rightarrow \mathbb{C}_3, s_3, ok$ and $\mathbb{C}_3, s_3 \xrightarrow{k} \mathbb{C}', s', er$. As such, from the inductive hypothesis, $\mathbb{C}_3, s_3 \xrightarrow{k} \mathbb{C}', s', er$ and $\mathbb{C}_2, s' \rightsquigarrow^+_{\text{er}} \text{skip}, s''$ we know $\mathbb{C}_3; \mathbb{C}_2, s_3 \xrightarrow{k} \mathsf{C}'; \mathsf{C}_2, s'', er$. Moreover, as $\mathbb{C}_1, s \rightarrow \mathbb{C}_3, s_3, ok$, from S-Seq1 we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_3; \mathbb{C}_2, s_3, ok$. Consequently, as $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_3; \mathbb{C}_2, s_3, ok$ and $\mathbb{C}_3; \mathbb{C}_2, s_3 \xrightarrow{k} \mathbb{C}'; \mathbb{C}_2, s'', er$, from the definition of $\xrightarrow{k+1}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \xrightarrow{k+1} \mathbb{C}'; \mathbb{C}_2, s'', er$, as required. □

Lemma 4. *For all* p, C, a, b, *if* $\forall n \in \mathbb{N}.\ a \leq n < b \Rightarrow \models_{\mathsf{B}} [p(n)]\ \mathsf{C}\ [ok\colon p(n{+}1)]$, *then* $\forall k, i \in \mathbb{N}.\ a \leq i \wedge i{+}k < b \Rightarrow \models_{\mathsf{B}} [p(i)]\ \mathsf{C}^\star\ [ok\colon p(i{+}k)]$.

Proof. Pick arbitrary p, C, a, b such that:

$$\forall n \in \mathbb{N}.\ a \leq n < b \Rightarrow \models_{\mathsf{B}} [p(n)]\ \mathsf{C}\ [ok\colon p(n{+}1)] \tag{1}$$

Pick an arbitrary k. We proceed by induction on k.

Base case $k{=}0$
Pick an arbitrary $i \in \mathbb{N}$ such that $a \leq i \wedge i{+}k < b$. We are then required to show $\models_{\mathsf{B}} [p(i)]\ \mathsf{C}^\star\ [ok\colon p(i)]$. Pick an arbitrary $s \in p(i)$. From S-Loop0 we have $\mathsf{C}^\star, s \rightarrow \text{skip}, s, ok$. As such, as we have $\text{skip}, s \xrightarrow{0} \text{skip}, s, ok$ (from the definition of $\xrightarrow{0}$), by definition we have $\mathsf{C}^\star, s \xrightarrow{1} \text{skip}, s, ok$. Consequently, we have $s \in p(i)$ and $\mathsf{C}^\star, s \xrightarrow{1} \text{skip}, s, ok$, as required.

Inductive case $k{=}j{+}1$

$$\forall c \in \mathbb{N}.\ a \leq c \wedge c{+}j < b \Rightarrow \models_{\mathsf{B}} [p(c)]\ \mathsf{C}^\star\ [ok\colon p(c{+}j)] \tag{I.H}$$

Pick an arbitrary $i \in \mathbb{N}$ such that $a \leq i \wedge i{+}k < b$. Pick an arbitrary $s \in p(i)$. As $a \leq i \wedge i{+}k < b$, from (1) we know $\models_{\mathsf{B}} [p(i)]\ \mathsf{C}\ [ok\colon p(i{+}1)]$ holds, and thus since $s \in p(i)$, from the definition of $\models_{\mathsf{B}}$ we then know there exists $s'' \in p(i{+}1), m$ such that $\mathsf{C}, s \xrightarrow{m} -, s'', ok$.

On the other hand, as $a \leq i$, $i{+}k < b$ and $k = j{+}1$ we also have $a \leq i{+}1$ and $i{+}j{+}1 < b$, i.e. $a \leq i{+}1$ and $(i{+}1){+}j < b$. As such, from the inductive hypothesis we have $\models_{\mathsf{B}} [p(i{+}1)]\ \mathsf{C}^\star\ [ok\colon p((i{+}1){+}j)]$, i.e. $\models_{\mathsf{B}} [p(i{+}1)]\ \mathsf{C}^\star\ [ok\colon p(i{+}k)]$. Consequently, since $s'' \in p(i{+}1)$, from the definition of $\models_{\mathsf{B}}$ we know there exists $s' \in p(i{+}k), d$ such that $\mathsf{C}^\star, s'', \xrightarrow{d} -, s', ok$. Therefore, from Lemma 2, $\mathsf{C}, s \xrightarrow{m} -, s'', ok$ and $\mathsf{C}^\star, s'', \xrightarrow{d} -, s', ok$ we know there exists e such that $\mathsf{C}; \mathsf{C}^\star, s, \xrightarrow{e} -, s', ok$.

Furthermore, from S-Loop we simply have $\mathsf{C}^\star, s, \rightarrow \mathsf{C}; \mathsf{C}^\star, s, ok$. As such, since we also have $\mathsf{C}; \mathsf{C}^\star, s, \xrightarrow{e} -, s', ok$, from the definition of $\xrightarrow{e+1}$ we have $\mathsf{C}^\star, s, \xrightarrow{e+1} -, s', ok$. That is, we have $s' \in p(i{+}k)$ such that $\mathsf{C}^\star, s, \xrightarrow{e+1} -, s', ok$, as required. □

Lemma 5 (BUA soundness). *For all* $p, \mathsf{C}, q, \epsilon$, *if* $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ *can be proven using the proof rules in Fig. 1, then* $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ *holds.*

Proof. By induction on the structure of rules in Fig. 1.

Case Skip
Pick arbitrary p such that $\vdash_B \left[p\right]$ skip $\left[ok\colon p\right]$. Pick an arbitrary $s \in p$. From the semantics of skip we then have skip, $s \xrightarrow{0}$ skip, s, ok and $s \in p$, as required.

Case Assign
Pick arbitrary p such that $\vdash_B \left[p\right]$ $x := e$ $\left[ok\colon \exists y.\ p[y/x] \wedge x = e[y/x]\right]$. 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 $x := e, s \longrightarrow$ skip, s', ok. As such, since we also have skip, $s' \xrightarrow{0}$ skip, s', ok, by definition we have $x := e, s \xrightarrow{1}$ 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 $s' \in p[v_x/x] \wedge x = e[v_x/x]$. Let $s'' = s'[y \mapsto v_x]$. Consequently, as $s' \in p[v_x/x] \wedge x = e[v_x/x]$, we also have $s'' \in p[y/x] \wedge x = e[y/x]$. As such, since $s'' \in p[y/x] \wedge x = e[y/x]$ and $s'' = s'[y \mapsto v_x]$, by definition we have $s' \in \exists y.\ p[y/x] \wedge x = e[y/x]$. Therefore, we have $x := e, s \xrightarrow{1}$ skip, s', ok and $s' \in \exists y.\ p[y/x] \wedge x = e[y/x]$, as required.

Case Assume
Pick arbitrary p, B such that $\vdash_B \left[p \wedge B\right]$ assume(B) $\left[ok\colon p \wedge B\right]$. Pick an arbitrary $s \in p \wedge B$. By definition we then know $s(B) =$ true. From S-Assume we then have assume$(B), s \longrightarrow$ skip, s, ok. As such, since we also have skip, $s \xrightarrow{0}$ skip, s, ok, by definition we have assume$(B), s \xrightarrow{1}$ skip, s, ok. Consequently, we have $s \in p \wedge B$ and assume$(B), s \xrightarrow{1}$ skip, s, ok, as required.

Case Error
Pick arbitrary p such that $\vdash_B \left[p\right]$ error $\left[er\colon p\right]$. Pick an arbitrary $s \in p$. From S-Error we then have error, $s \longrightarrow$ skip, s, er. As such, by definition we have error, $s \xrightarrow{1}$ skip, s, er. Consequently, we have $s \in p$ and error, $s \xrightarrow{1}$ skip, s, er, as required.

Case Seq
Pick arbitrary $p, q, r, C_1, C_2, \epsilon$ such that $\vdash_B \left[p\right]$ C_1 $\left[ok\colon r\right]$ and $\vdash_B \left[r\right]$ C_2 $\left[\epsilon :q\right]$. Pick an arbitrary $s \in p$. From $\vdash_B \left[p\right]$ C_1 $\left[ok\colon r\right]$ and the inductive hypothesis we then know there exists $s'' \in r, i$ such that $C_1, s \xrightarrow{i} -, s'', ok$. Moreover, as $s'' \in r, i$, from $\vdash_B \left[r\right]$ C_2 $\left[\epsilon :q\right]$ and the inductive hypothesis we know there exists $s' \in q, j$ such that $C_2, s'' \xrightarrow{j} -, s', \epsilon$. As $C_1, s \xrightarrow{i} -, s'', ok$ and $C_2, s'' \xrightarrow{j} -, s', \epsilon$, from Lemma 2 we know there exists n such that $C_1; C_2, s \xrightarrow{n} -, s', \epsilon$. That is, there exists $s' \in q, n$ such that $C_1; C_2, s \xrightarrow{n} -, s', \epsilon$, as required.

Case SeqEr
Pick arbitrary p, q, C_1, C_2 such that $\vdash_B \left[p\right]$ $C_1; C_2$ $\left[er\colon q\right]$. Pick an arbitrary $s \in p$. From the $\vdash_B \left[p\right]$ C_1 $\left[er\colon q\right]$ premise and the inductive hypothesis we then know there exists $s' \in q, i$ such that $C_1, s \xrightarrow{i} -, s', er$. From Prop. 2 we know $C_2, s' \rightsquigarrow^{+}_{\mathrm{er}} s'$, and thus from $C_1, s \xrightarrow{i} -, s', er$ and Lemma 3 we know $C_1; C_2, s \xrightarrow{i} -, s', er$. That is, there exists $s' \in q$ such that $C_1; C_2, s \xrightarrow{i} -, s', er$, as required.

Case Choice
Pick arbitrary $p, q, \mathsf{C}_1, \mathsf{C}_2, \epsilon$ and $i \in \{1,2\}$ such that $\vdash_{\mathrm{B}} [p]\ \mathsf{C}_1 + \mathsf{C}_2\ [\epsilon : q]$. Pick an arbitrary $s \in p$. From the $\vdash_{\mathrm{B}} [p]\ \mathsf{C}_i\ [\epsilon : q]$ premise and the inductive hypothesis we then know there exists $s' \in q, j$ such that $\mathsf{C}_i, s \xrightarrow{j} -, s', \epsilon$. Moreover, from S-Choice we have $\mathsf{C}_1 + \mathsf{C}_2, s \rightarrow \mathsf{C}_i, s, ok$. As such, from the definition of $\xrightarrow{j+1}$ we have $\mathsf{C}_1 + \mathsf{C}_2, s \xrightarrow{j+1} -, s', \epsilon$. That is, there exists $s' \in q$ such that $\mathsf{C}_1 + \mathsf{C}_2, s \xrightarrow{j+1} -, s', \epsilon$, as required.

Case Loop0
Pick arbitrary p, C such that $\vdash_{\mathrm{B}} [p]\ \mathsf{C}^{\star}\ [ok : p]$. Pick an arbitrary $s \in p$. From S-Loop0 we have $\mathsf{C}^{\star}, s \rightarrow \mathsf{skip}, s, ok$. As such, as we have $\mathsf{skip}, s \xrightarrow{0} \mathsf{skip}, s, ok$ (from the definition of $\xrightarrow{0}$), by definition we have $\mathsf{C}^{\star}, s \xrightarrow{1} \mathsf{skip}, s, ok$. Consequently, we have $s \in p$ and $\mathsf{C}^{\star}, s \xrightarrow{1} \mathsf{skip}, s, ok$, as required.

Case Loop
Pick arbitrary p, C, q such that $\vdash_{\mathrm{B}} [p]\ \mathsf{C}^{\star}\ [\epsilon : q]$. Pick an arbitrary $s \in p$. From the $\vdash_{\mathrm{B}} [p]\ \mathsf{C}^{\star}; \mathsf{C}\ [\epsilon : q]$ premise and the inductive hypothesis we know there exists $s' \in q, j$ such that $\mathsf{C}^{\star}; \mathsf{C}, s \xrightarrow{j} -, s', \epsilon$. From Prop. 3 we then know there exists i such that $\mathsf{C}; \mathsf{C}^{\star}, s \xrightarrow{i} -, s', \epsilon$. From S-Loop we have $\mathsf{C}^{\star}, s \rightarrow \mathsf{C}; \mathsf{C}^{\star}, s, ok$. As such, from the definition of $\xrightarrow{i+1}$ we have $\mathsf{C}^{\star}, s \xrightarrow{i+1} -, s', \epsilon$. Consequently, we have $s \in p$ and $\mathsf{C}^{\star}, s \xrightarrow{i+1} -, s', \epsilon$, as required.

Case Loop-Subvar
Pick arbitrary p, C, k such that $\vdash_{\mathrm{B}} [p(0)]\ \mathsf{C}^{\star}\ [ok : p(k)]$. From the $\forall n < k.\ \vdash_{\mathrm{B}} [p(n)]\ \mathsf{C}\ [ok : p(n{+}1)]$ premise and the inductive hypothesis we have $\forall n < k.\ \models_{\mathrm{B}} [p(n)]\ \mathsf{C}\ [ok : p(n{+}1)]$. Consequently, from Lemma 4 we have $\models_{\mathrm{B}} [p(0)]\ \mathsf{C}^{\star}\ [ok : p(k)]$, as required.

Case Local
Pick arbitrary p, C, q, ϵ such that $\vdash_{\mathrm{B}} [\exists x.\ p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\epsilon : \exists x.\ q]$. 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 $\vdash_{\mathrm{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ premise and the inductive hypothesis we know there exists $s_q \in q$ and n such that $\mathsf{C}, s_p \xrightarrow{n} -, s_q, \epsilon$. From S-Local we have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, s \rightarrow \mathsf{C}; \mathsf{end}(x, s(x)), s_p$. There are now two cases to consider: 1) $\epsilon{=}ok$; or 2) $\epsilon{=}er$.

In case (1), let $s'' = s_q[x \mapsto s(x)]$. From S-LocalEnd we then have $\mathsf{end}(x, s(x)), s_q \rightarrow \mathsf{skip}, s''$. From the definition of $\xrightarrow{0}$ we have $\mathsf{skip}, s'' \xrightarrow{0} \mathsf{skip}, s'', ok$, and thus since we have $\mathsf{end}(x, s(x)), s_q \rightarrow \mathsf{skip}, s''$, from the definition of $\xrightarrow{1}$ we have $\mathsf{end}(x, s(x)), s_q \xrightarrow{1} \mathsf{skip}, s''$. Consequently, since we also have $\mathsf{C}, s_p \xrightarrow{n} -, s_q, \epsilon$, from Lemma 2 we know there exists m such that $\mathsf{C}; \mathsf{end}(x, s(x)), s_p \xrightarrow{m} \mathsf{skip}, s'', ok$. On the other hand, since we have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, s \rightarrow \mathsf{C}; \mathsf{end}(x, s(x)), s_p$, by definition of $\xrightarrow{m+1}$ we also have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, s \xrightarrow{m+1} \mathsf{skip}, s'', ok$. 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), let $s'' = s_q[x \mapsto s(x)]$. From $\rightsquigarrow_{\mathrm{er}}$ transitions we then have $\mathsf{end}(x, s(x)), s_q \rightsquigarrow_{\mathrm{er}} \mathsf{skip}, s''$. As such, from $\mathsf{C}, s_p \xrightarrow{n} -, s_q, \epsilon$ and Lemma 3 we have $\mathsf{C}; \mathsf{end}(x, s(x)), s_p \xrightarrow{n} -, s'', \epsilon$. On the other hand, since we have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, s \rightarrow \mathsf{C}; \mathsf{end}(x, s(x)), s_p$, by definition of $\xrightarrow{n+1}$ we also have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, s \xrightarrow{n+1} -, s'', \epsilon$. 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.

Case Subst
Pick arbitrary p, C, q, y such that $y \notin \mathrm{fv}(p, \mathrm{C}, q)$ and $(\vdash_\mathrm{B} [p]\ \mathrm{C}\ [\epsilon : q])[y/x]$, i.e. $\vdash_\mathrm{B} [p[y/x]]\ \mathrm{C}[y/x]\ [\epsilon : q[y/x]]$. 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 $\vdash_\mathrm{B} [p]\ \mathrm{C}\ [\epsilon : q]$ premise and the inductive hypothesis we know there exists $s_q \in q, n$ such that $\mathrm{C}, s_p \xrightarrow{n} -, s_q \epsilon$. 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 $\mathrm{C}[y/x], s \xrightarrow{n} -, s', \epsilon$, as required.

Case Disj
Pick arbitrary I such that $\vdash_\mathrm{B} [p_i]\ \mathrm{C}\ [\epsilon : q_i]$ for all $i \in I$. Pick an arbitrary $s \in \bigvee_{i \in I} p_i$. We then know there exists $j \in I$ such that $s \in p_j$. From the $\vdash_\mathrm{B} [p_j]\ \mathrm{C}\ [\epsilon : q_j]$ premise and the inductive hypothesis we know there exists $s' \in q_j, n$ such that $\mathrm{C}, s \xrightarrow{n} -, s', \epsilon$. That is, there exists $s' \in \bigvee_{i \in I} q_i$ and n such that $\mathrm{C}, s \xrightarrow{n} -, s', \epsilon$, as required.

Case Constancy
Pick arbitrary p, q, r, C such that $\vdash_\mathrm{B} [p \wedge r]\ \mathrm{C}\ [\epsilon : q \wedge r]$. Pick an arbitrary $s \in p \wedge r$. That is, $s \in p$ and $s \in r$. From the $\vdash_\mathrm{B} [p]\ \mathrm{C}\ [\epsilon : q]$ premise and the inductive hypothesis we know there exists $s' \in q, n$ such that $\mathrm{C}, s \xrightarrow{n} -, s', \epsilon$. As such, from the $\mathrm{fv}(r) \cap \mathrm{mod}(\mathrm{C}) = \emptyset$ premise, Prop. 1 and since $s \in r$, we know $s' \in r$. Therefore, we have $s' \in q$ and $s' \in r$ and thus $s' \in q \wedge r$. That is, there exists $s' \in q \wedge r$ and n such that $\mathrm{C}, s \xrightarrow{n} -, s', \epsilon$, as required.

Case ConsB
Pick arbitrary p, q, C such that $\vdash_\mathrm{B} [p]\ \mathrm{C}\ [\epsilon : q]$. Pick an arbitrary $s \in p$. From the $p \subseteq p'$ premise we then have $s \in p'$. Moreover, from the $\vdash_\mathrm{B} [p']\ \mathrm{C}\ [\epsilon : q']$ and the inductive hypothesis we know there exists $s' \in q'$ and n such that $\mathrm{C}, s \xrightarrow{n} -, s', \epsilon$. 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 $\mathrm{C}, s \xrightarrow{n} -, s', \epsilon$, as required.

Case DisjTrack
Pick arbitrary $P_1, P_2, Q_1, Q_2, \mathrm{C}$ such that $\vdash_\mathrm{B} [P_1 \uplus P_2]\ \mathrm{C}\ [\epsilon : Q_1 \uplus Q_2]$. Pick an arbitrary $i \in dom(P_1 \uplus P_2)$ and $s \in (P_1 \uplus P_2)(i)$. We then know that either $i \in dom(P_1)$ or $i \in dom(P_2)$. Without loss of generality, let us assume $i \in dom(P_1)$.

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

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

Lemma 6 (FUA soundness). *For all p, C, q, ϵ, if $\vdash_\mathrm{F} [p]\ \mathrm{C}\ [\epsilon : q]$ can be proven using the proof rules in Fig. 1, then $\models_\mathrm{F} [p]\ \mathrm{C}\ [\epsilon : q]$ holds.*

Proof. By induction on the structure of rules in Fig. 1.

Cases Skip, Assign, Error, Seq, SeqEr, Choice, Loop0, Loop, Loop-Subvar, Disj, Constancy, ConsF, Subst, Local
The proof of these cases is as given by O'Hearn [2019].

Case Assume
Pick arbitrary p, B such that $\vdash_{\mathsf{F}} \big[p \wedge B\big]$ assume(B) $\big[ok\colon p \wedge B\big]$. Pick an arbitrary $s \in p \wedge B$. By definition we then know $s(B)$ = true. From S-Assume we then have assume$(B), s \longrightarrow$ skip, s, ok. As such, since we also have skip, $s \xrightarrow{0}$ skip, s, ok, by definition we have assume$(B), s \xrightarrow{1}$ skip, s, ok. Consequently, we have $s \in p \wedge B$ and assume$(B), s \xrightarrow{1}$ skip, s, ok, as required.

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

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

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

Theorem 13 (Soundness). *For all p,* C*, q, ϵ, if* $\vdash_{\dagger}$ $[p]$ C $[\epsilon : q]$ *can be proven using the proof rules in Fig. 1, then* $\models_{\dagger}$ $[p]$ C $[\epsilon : q]$ *holds.*

Proof. Follows immediately from Lemma 5 and Lemma 6. □

C.2 Soundness of Divergence Rules

In what follows, we write C, $s \rightsquigarrow^{+}$ C$'$, s', ϵ for $\exists n$. C, $s \rightsquigarrow^{n}$ C$'$, s', ϵ.

Lemma 7. *For all $\mathbb{C}, s, \mathbb{C}', s', \epsilon, n$, if $n > 0$ and $\mathbb{C}, s \xrightarrow{n} \mathbb{C}', s', \epsilon$, then $\mathbb{C}, s \rightsquigarrow^{n} \mathbb{C}', s', \epsilon$.*

Proof. By induction on n.

Base case $n = 1$
Pick arbitrary $\mathbb{C}, \mathbb{C}', s, \mathbb{C}', s', \epsilon$ such that $\mathbb{C}, s \xrightarrow{1} \mathbb{C}', s', \epsilon$. From the definition of $\xrightarrow{1}$ there are then two cases to consider: 1) ϵ=er and there exists s'' such that $\mathbb{C}, s \longrightarrow \mathbb{C}', s''$, er and $\mathbb{C}', s'' \rightsquigarrow^{+}_{\mathsf{er}}$, skip, s'; or 2) ϵ=ok, $\mathbb{C}'$ = skip and $\mathbb{C}, s \longrightarrow \mathbb{C}', s'$, ok.

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

Inductive case $n = k{+}1$ **with** $k > 0$
Pick arbitrary $\mathbb{C}, \mathbb{C}', s, \mathbb{C}', s', \epsilon$ such that $\mathbb{C}, s \xrightarrow{n} \mathbb{C}', s', \epsilon$. From the definition of $\xrightarrow{n}$, we know there exists $\mathbb{C}'', s''$ such that $\mathbb{C}, s \rightarrow \mathbb{C}'', s'', \mathit{ok}$ and $\mathbb{C}'', s'' \xrightarrow{k} \mathbb{C}', s', \epsilon$. From $\mathbb{C}'', s'' \xrightarrow{k} \mathbb{C}', s', \epsilon$ and the inductive hypothesis we have $\mathbb{C}'', s'' \rightsquigarrow^{k} \mathbb{C}', s', \epsilon$. As such, from $\mathbb{C}, s \rightarrow \mathbb{C}'', s'', \mathit{ok}$ and the definition of $\rightsquigarrow^{n}$ we have $\mathbb{C}, s \rightsquigarrow^{n} \mathbb{C}', s', \epsilon$, as required. □

Lemma 8. *For all* $n, \mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_1', s, \mathbb{C}', s'$, *if* $\mathbb{C}_1, s \rightsquigarrow^{n} \mathbb{C}_1', s', \mathit{ok}$, *then* $\mathbb{C}_1; \mathbb{C}_2, s \rightsquigarrow^{n} \mathbb{C}_1'; \mathbb{C}_2, s', \mathit{ok}$.

Proof. By induction on n.

Base case $n = 1$
Pick arbitrary $\mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_1', s, \mathbb{C}', s'$ such that $\mathbb{C}_1, s \rightsquigarrow^{1} \mathbb{C}_1', s', \mathit{ok}$. From the definition of $\rightsquigarrow^{1}$ we then know $\mathbb{C}_1, s \rightarrow \mathbb{C}_1', s', \mathit{ok}$. From S-Seq1 we then have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_1'; \mathbb{C}_2, s', \epsilon$, and thus by definition of $\rightsquigarrow^{1}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \rightsquigarrow^{1} \mathbb{C}_1'; \mathbb{C}_2, s', \epsilon$, as required.

Inductive case $n = k{+}1$
Pick arbitrary $\mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_1', s, \mathbb{C}', s'$ such that $\mathbb{C}_1, s \rightsquigarrow^{n} \mathbb{C}_1', s', \mathit{ok}$. From the definition of $\rightsquigarrow^{n}$ we then know there exists $\mathbb{C}'', s''$ such that $\mathbb{C}_1, s \rightarrow \mathbb{C}'', s'', \mathit{ok}$ and $\mathbb{C}'', s'' \rightsquigarrow^{k} \mathbb{C}_1', s', \mathit{ok}$. From $\mathbb{C}_1, s \rightarrow \mathbb{C}'', s'', \mathit{ok}$ and S-Seq1 we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}''; \mathbb{C}_2, s'', \mathit{ok}$. From $\mathbb{C}'', s'' \rightsquigarrow^{k} \mathbb{C}_1', s', \mathit{ok}$ and the inductive hypothesis we have $\mathbb{C}''; \mathbb{C}_2, s'' \rightsquigarrow^{k} \mathbb{C}_1'; \mathbb{C}_2, s', \mathit{ok}$. As such, since we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}''; \mathbb{C}_2, s'', \mathit{ok}$ and $\mathbb{C}''; \mathbb{C}_2, s'' \rightsquigarrow^{k} \mathbb{C}_1'; \mathbb{C}_2, s', \mathit{ok}$, from the definition of $\rightsquigarrow^{n}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \rightsquigarrow^{n} \mathbb{C}_1'; \mathbb{C}_2, s', \mathit{ok}$, as required. □

Lemma 9. *For all* $s, s', s'', \mathbb{C}_1, \mathbb{C}_2, \mathbb{C}', i, j, \epsilon$, *if* $\mathbb{C}_1, s \xrightarrow{i} -, s'', \mathit{ok}$ *and* $\mathbb{C}_2, s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$, *then there exists* n *such that* $\mathbb{C}_1; \mathbb{C}_2, s \rightsquigarrow^{n} \mathbb{C}', s', \epsilon$.

Proof. Pick arbitrary $s, s', s'', \mathbb{C}_1, \mathbb{C}_2, \mathbb{C}', \mathbb{C}'', i, j, \epsilon$, such that $\mathbb{C}_1, s \xrightarrow{i} \mathbb{C}'', s'', \mathit{ok}$ and $\mathbb{C}_2, s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$. We proceed by induction on i.

Case $i = 0$
From $\mathbb{C}_1, s \xrightarrow{0} \mathbb{C}'', s'', \mathit{ok}$ we know $\mathbb{C}_1 = \mathbb{C}'' = \mathtt{skip}$ and $s = s''$. As such, since $\mathbb{C}_1 = \mathtt{skip}$ and $s = s''$, from S-SeqSkip we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_2, s'', \mathit{ok}$. Consequently, from $\mathbb{C}_2, s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$ and the definition of $\rightsquigarrow^{j+1}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \rightsquigarrow^{j+1} \mathbb{C}', s', \epsilon$, as required.

Case $i = k{+}1$
From the definition of $\mathbb{C}_1, s \xrightarrow{i} \mathbb{C}'', s'', \mathit{ok}$ we then know there exists $\mathbb{C}_3, s_3$ such that $\mathbb{C}_1, s \rightarrow \mathbb{C}_3, s_3, \mathit{ok}$ and $\mathbb{C}_3, s_3 \xrightarrow{k} \mathbb{C}'', s'', \mathit{ok}$. As such, from the inductive hypothesis, $\mathbb{C}_3, s_3 \xrightarrow{k} \mathbb{C}'', s'', \mathit{ok}$ and $\mathbb{C}_2, s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$ we know there exists n such that $\mathbb{C}_3; \mathbb{C}_2, s_3 \rightsquigarrow^{n} \mathbb{C}', s', \epsilon$. Moreover, as $\mathbb{C}_1, s \rightarrow \mathbb{C}_3, s_3, \mathit{ok}$, from S-Seq1 we have $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_3; \mathbb{C}_2, s_3, \mathit{ok}$. Consequently, as $\mathbb{C}_1; \mathbb{C}_2, s \rightarrow \mathbb{C}_3; \mathbb{C}_2, s_3, \mathit{ok}$ and $\mathbb{C}_3; \mathbb{C}_2, s_3 \rightsquigarrow^{n} \mathbb{C}', s', \epsilon$, from the definition of $\rightsquigarrow^{n+1}$ we have $\mathbb{C}_1; \mathbb{C}_2, s \rightsquigarrow^{n+1} \mathbb{C}', s', \epsilon$, as required. □

Lemma 10. *For all* $i, j, \mathbb{C}, \mathbb{C}', \mathbb{C}'', s, s', s'', \epsilon$, *if* $\mathbb{C}, s \rightsquigarrow^{i} \mathbb{C}'', s'', \mathit{ok}$ *and* $\mathbb{C}'', s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$, *then* $\mathbb{C}, s \rightsquigarrow^{i+j} \mathbb{C}', s', \epsilon$.

Proof. By induction on i.

Base case $i{=}1$
Pick arbitrary $j, \mathbb{C}, \mathbb{C}', \mathbb{C}'', s, s', s'', \epsilon$ such that $\mathbb{C}, s \rightsquigarrow^{1} \mathbb{C}'', s'', \mathit{ok}$ and $\mathbb{C}'', s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$. From $\mathbb{C}, s \rightsquigarrow^{1} \mathbb{C}'', s'', \mathit{ok}$ we then know $\mathbb{C}, s \rightarrow \mathbb{C}'', s'', \mathit{ok}$, and thus from $\mathbb{C}'', s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$ and the

definition of $\rightsquigarrow^{j+1}$ we have $\mathbb{C}, s \rightsquigarrow^{j+1} \mathbb{C}', s', \epsilon$, as required.

Inductive case $i{=}k{+}1$ **and** $k > 0$
Pick arbitrary $j, \mathbb{C}, \mathbb{C}', \mathbb{C}'', s, s', s'', \epsilon$ such that $\mathbb{C}, s \rightsquigarrow^{i} \mathbb{C}'', s'', ok$ and $\mathbb{C}'', s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$. From $\mathbb{C}, s \rightsquigarrow^{i} \mathbb{C}'', s'', ok$ and the definition of $\rightsquigarrow^{i}$ we know there exists $\mathbb{C}''', s'''$ such that $\mathbb{C}, s \rightarrow \mathbb{C}''', s''', ok$, and $\mathbb{C}''', s''' \rightsquigarrow^{k} \mathbb{C}'', s'', ok$. Consequently, from $\mathbb{C}''', s''' \rightsquigarrow^{k} \mathbb{C}'', s'', ok$, $\mathbb{C}'', s'' \rightsquigarrow^{j} \mathbb{C}', s', \epsilon$ and the inductive hypothesis we have $\mathbb{C}''', s''' \rightsquigarrow^{k+j} \mathbb{C}', s', \epsilon$. As such, from $\mathbb{C}, s \rightarrow \mathbb{C}''', s''', ok$ and the definition of $\rightsquigarrow^{k+j+1}$ we have $\mathbb{C}, s \rightsquigarrow^{k+j+1} \mathbb{C}', s', \epsilon$. That is, $\mathbb{C}, s \rightsquigarrow^{i+j} \mathbb{C}', s', \epsilon$, as required. □

Theorem 14 (Divergence soundness). *For all p, C, if $\vdash [p]\ \mathsf{C}\ [\infty]$ can be proven using the proof rules in Fig. 2, then $\models [p]\ \mathsf{C}\ [\infty]$ holds.*

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

Case Div-Local
Pick arbitrary p, C such that $\vdash [\exists x.\ p]$ local x in $\mathsf{C}\ [\infty]$. 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 $\vdash [p]\ \mathsf{C}\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $s_1, s_2, \cdots$ such that $\mathsf{C}, s_p \rightsquigarrow^{+} \mathsf{C}_1, s_1, ok \rightsquigarrow^{+} \mathsf{C}_2, s_2, ok \rightsquigarrow^{+} \cdots$. As such, from the definition of $\rightsquigarrow^{+}$ and Lemma 8 we have $\mathsf{C}; \mathsf{end}(x, s(x)), s_p \rightsquigarrow^{+} \mathsf{C}_1; \mathsf{end}(x, s(x)), s_1, ok \rightsquigarrow^{+} \mathsf{C}_2; \mathsf{end}(x, s(x)), s_2, ok \rightsquigarrow^{+} \cdots$ On the other hand, from S-Local we have local x in $\mathsf{C}, s \longrightarrow \mathsf{C}; \mathsf{end}(x, s(x)), s_p$. Therefore, since we also have $\mathsf{C}; \mathsf{end}(x, s(x)), s_p \rightsquigarrow^{+} \mathsf{C}_1; \mathsf{end}(x, s(x)), s_1, ok \rightsquigarrow^{+} \mathsf{C}_2; \mathsf{end}(x, s(x)), s_2, ok \rightsquigarrow^{+} \cdots$, from the definition of $\rightsquigarrow^{+}$ we have local x in $\mathsf{C}, s \rightsquigarrow^{+} \mathsf{C}_1; \mathsf{end}(x, s(x)), s_1, ok \rightsquigarrow^{+} \mathsf{C}_2; \mathsf{end}(x, s(x)), s_2, ok \rightsquigarrow^{+} \cdots$, as required.

Case Div-Disj
Pick arbitrary I, p_i, C such that $\vdash [\bigvee_{i \in I} p_i]\ \mathsf{C}\ [\infty]$. Pick an arbitrary $s \in \bigvee_{i \in I} p_i$. We then know there exists $j \in I$ such that $s \in p_j$. From the $\vdash [p_j]\ \mathsf{C}\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $s_1, s_2, \cdots$ such that $\mathsf{C}, s \rightsquigarrow^{+} \mathsf{C}_1, s_1, ok \rightsquigarrow^{+} \mathsf{C}_2, s_2, ok \rightsquigarrow^{+} \cdots$, as required.

Case Div-Seq1
Pick arbitrary p, C_1, C_2 such that $[p]\ \mathsf{C}_1; \mathsf{C}_2\ [\infty]$. Pick an arbitrary $s \in p$. From the $[p]\ \mathsf{C}_1\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $\mathsf{C}'_2, \mathsf{C}'_3, \cdots$, and $s_2, s_3, \cdots$, such that $\mathsf{C}_1, s \rightsquigarrow^{+} \mathsf{C}'_2, s_2, ok \rightsquigarrow^{+} \mathsf{C}'_3, s_3, ok \rightsquigarrow^{+} \cdots$. As such, from the definition of $\rightsquigarrow^{+}$ and Lemma 8 we have $\mathsf{C}_1; \mathsf{C}_2, s \rightsquigarrow^{+} \mathsf{C}'_2; \mathsf{C}_2, s_2, ok \rightsquigarrow^{+} \mathsf{C}'_3; \mathsf{C}_2, s_3, ok \rightsquigarrow^{+} \cdots$, as required.

Case Div-Seq2
Pick arbitrary p, q, C_1, C_2 such that $[p]\ \mathsf{C}_1; \mathsf{C}_2\ [\infty]$. Pick an arbitrary $s \in p$. From the $\vdash_{\mathrm{B}} [p]\ \mathsf{C}_1\ [ok\colon q]$ premise and Theorem 13 we know there exists $s_q \in q$ and n such that $\mathsf{C}_1, s \xrightarrow{n} -, s_q, ok$. Moreover, from the $[q]\ \mathsf{C}_2\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $\mathsf{C}'_3, \mathsf{C}'_4, \cdots$ and $s_3, s_4, \cdots$, such that $\mathsf{C}_2, s_q \rightsquigarrow^{+} \mathsf{C}'_3, s_3, ok \rightsquigarrow^{+} \mathsf{C}'_4, s_4, ok \rightsquigarrow^{+} \cdots$. As $\mathsf{C}_1, s \xrightarrow{n} -, s_q, ok$ and $\mathsf{C}_2, s_q \rightsquigarrow^{+} \mathsf{C}'_3, s_3, ok$, from the definition of $\rightsquigarrow^{+}$ and Lemma 9 we have $\mathsf{C}_1; \mathsf{C}_2, s \rightsquigarrow^{+} \mathsf{C}'_3, s_3, ok$. Moreover, as $\mathsf{C}'_3, s_3 \rightsquigarrow^{+} \mathsf{C}'_4, s_4, ok \rightsquigarrow^{+} \cdots$, we have $\mathsf{C}_1; \mathsf{C}_2, s \rightsquigarrow^{+} \mathsf{C}'_3, s_3, ok \rightsquigarrow^{+} \mathsf{C}'_4, s_4, ok \rightsquigarrow^{+} \cdots$, as required.

Case Div-Choice
Pick arbitrary p, $\mathsf{C}_1, \mathsf{C}_2$ such that $[p]\ \mathsf{C}_1 + \mathsf{C}_2\ [\infty]$. Pick an arbitrary $s \in p$ and $i \in \{1, 2\}$. From the $[p]\ \mathsf{C}_i\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $\mathsf{C}_3, \mathsf{C}_4, \cdots$ and $s_3, s_4, \cdots$, such that $\mathsf{C}_i, s \rightsquigarrow^+ \mathsf{C}_3, s_3, ok \rightsquigarrow^+ \mathsf{C}_4, s_4, ok \rightsquigarrow^+ \cdots$. Moreover, from S-Choice we have $\mathsf{C}_1 + \mathsf{C}_2, s \rightarrow \mathsf{C}_i, s, ok$. And thus we have $\mathsf{C}_1 + \mathsf{C}_2, s \rightarrow \mathsf{C}_i, s, ok \rightsquigarrow^+ \mathsf{C}_3, s_3, ok \rightsquigarrow^+ \mathsf{C}_4, s_4, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-LoopUnfold
Pick arbitrary p, C such that $[p]\ \mathsf{C}^\star\ [\infty]$. Pick an arbitrary $s \in p$. From the $[p]\ \mathsf{C};\mathsf{C}^\star\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $s_1, s_2, \cdots$, such that $\mathsf{C};\mathsf{C}^\star, s \rightsquigarrow^+ \mathsf{C}_1, s_1, ok \rightsquigarrow^+ \mathsf{C}_2, s_2, ok \rightsquigarrow^+ \cdots$. Moreover, from S-Loop we have $\mathsf{C}^\star, s \rightarrow \mathsf{C};\mathsf{C}^\star, s, ok$. And thus we have $\mathsf{C}^\star, s \rightarrow \mathsf{C};\mathsf{C}^\star, s, ok \rightsquigarrow^+ \mathsf{C}_1, s_1, ok \rightsquigarrow^+ \mathsf{C}_2, s_2, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-LoopNest
This rule can be derived as follows:

$$\dfrac{\dfrac{[p]\ \mathsf{C}\ [\infty]}{[p]\ \mathsf{C};\mathsf{C}^\star\ [\infty]}\ \textsc{Div-Seq1}}{[p]\ \mathsf{C}^\star\ [\infty]}\ \textsc{Div-LoopUnfold}$$

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 $\vdash [p]\ \mathsf{C}^\star\ [\infty]$. From S-Loop we then have:

$$\forall s \in p.\ \mathsf{C}^\star, s \rightarrow \mathsf{C};\mathsf{C}^\star, s, ok \tag{2}$$

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

$$\forall s \in p.\ \mathsf{C}, s \xrightarrow{g(s)} \mathsf{skip}, f(s), ok \wedge f(s) \in p \tag{3}$$

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

Pick an arbitrary $s \in p$. From (3) we have $\mathsf{C}, s \xrightarrow{g(s)} \mathsf{skip}, f(s), ok$. There are now two cases to consider: i) $g(s) = 0$; or ii) $g(s) > 0$. In case (i), we then have $\mathsf{C} = \mathsf{skip}$ and $s{=}f(s)$. As such, from S-SeqSkip we have $\mathsf{C};\mathsf{C}^\star, s \rightarrow \mathsf{C}^\star, f(s), ok$, and thus by definition of $\rightsquigarrow^1$ we have $\mathsf{C};\mathsf{C}^\star, s \rightsquigarrow^1 \mathsf{C}^\star, f(s), ok$

In case (ii), from $\mathsf{C}, s \xrightarrow{g(s)} \mathsf{skip}, f(s), ok$ and Lemma 7 we have $\mathsf{C}, s \rightsquigarrow^{g(s)} \mathsf{skip}, f(s), ok$. Consequently, from Lemma 8 we have $\mathsf{C};\mathsf{C}^\star, s \rightsquigarrow^{g(s)} \mathsf{skip};\mathsf{C}^\star, f(s), ok$. On the other hand, from S-SeqSkip we have $\mathsf{skip};\mathsf{C}^\star, f(s) \rightarrow \mathsf{C}^\star, f(s), ok$ and thus by definition of $\rightsquigarrow^1$ we have $\mathsf{skip};\mathsf{C}^\star, f(s) \rightsquigarrow^1 \mathsf{C}^\star, f(s), ok$. From Lemma 10, $\mathsf{C};\mathsf{C}^\star, s \rightsquigarrow^{g(s)} \mathsf{skip};\mathsf{C}^\star, f(s), ok$ and $\mathsf{skip};\mathsf{C}^\star, f(s) \rightsquigarrow^1 \mathsf{C}^\star, f(s), ok$ we know there exists i such that $\mathsf{C};\mathsf{C}^\star, s \rightsquigarrow^i \mathsf{C}^\star, f(s), ok$.

That is, in both cases we know there exists i such that $\mathsf{C};\mathsf{C}^\star, s \rightsquigarrow^i \mathsf{C}^\star, f(s), ok$. As such, from (2) and the definition of $\rightsquigarrow^{i+1}$ we have $\mathsf{C}^\star, s \rightsquigarrow^{i+1} \mathsf{C}^\star, f(s), ok$, i.e. $\mathsf{C}^\star, s \rightsquigarrow^+ \mathsf{C}^\star, f(s), ok$. That is, from (3) we have:

$$\forall s \in p.\ \mathsf{C}^\star, s \rightsquigarrow^+ \mathsf{C}^\star, f(s), ok \wedge f(s) \in p \tag{4}$$

Pick an arbitrary $s \in p$. From (4) we then know $\mathsf{C}^\star, s \rightsquigarrow^+ \mathsf{C}^\star, f(s), ok \rightsquigarrow^+ \mathsf{C}^\star, f^2(s), ok \rightsquigarrow^+ \cdots$, as required.

Case Div-Subvar

Pick arbitrary p, C, q such that $\vdash [p(0)]\ \mathsf{C}^\star\ [\infty]$. From S-Loop we then have:

$$\forall s \in p(0).\ \mathsf{C}^\star, s \rightarrow \mathsf{C}; \mathsf{C}^\star, s, ok \tag{5}$$

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

$$\forall i \in \mathbb{N}^+.\ \forall s \in p(i-1).\ \mathsf{C}, s \xrightarrow{g_i(s)} \mathsf{skip}, f_i(s), ok \land f_i(s) \in p(i) \tag{6}$$

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

Pick an arbitrary $i \in \mathbb{N}^+$ and $s \in p(i{-}1)$. From (6) we have $\mathsf{C}, s \xrightarrow{g_i(s)} \mathsf{skip}, f_i(s), ok$. There are now two cases to consider: a) $g_i(s) = 0$; or b) $g_i(s) > 0$. In case (a), we then have $\mathsf{C} = \mathsf{skip}$ and $s{=}f_i(s)$. As such, from S-SeqSkip we have $\mathsf{C}; \mathsf{C}^\star, s \rightarrow \mathsf{C}^\star, f_i(s), ok$, and thus by definition of $\rightsquigarrow^1$ we have $\mathsf{C}; \mathsf{C}^\star, s \rightsquigarrow^1 \mathsf{C}^\star, f_i(s), ok$

In case (b), from $\mathsf{C}, s \xrightarrow{g_i(s)} \mathsf{skip}, f_i(s), ok$ and Lemma 7 we have $\mathsf{C}, s \rightsquigarrow^{g_i(s)} \mathsf{skip}, f_i(s), ok$. Consequently, from Lemma 8 we have $\mathsf{C}; \mathsf{C}^\star, s \rightsquigarrow^{g_i(s)} \mathsf{skip}; \mathsf{C}^\star, f_i(s), ok$. On the other hand, from S-SeqSkip we have $\mathsf{skip}; \mathsf{C}^\star, f_i(s) \rightarrow \mathsf{C}^\star, f_i(s), ok$ and thus by definition of $\rightsquigarrow^1$ we have $\mathsf{skip}; \mathsf{C}^\star, f_i(s) \rightsquigarrow^1 \mathsf{C}^\star, f_i(s), ok$. From Lemma 10, $\mathsf{C}; \mathsf{C}^\star, s \rightsquigarrow^{g_i(s)} \mathsf{skip}; \mathsf{C}^\star, f_i(s), ok$ and $\mathsf{skip}; \mathsf{C}^\star, f_i(s) \rightsquigarrow^1 \mathsf{C}^\star, f_i(s), ok$ we know there exists j such that $\mathsf{C}; \mathsf{C}^\star, s \rightsquigarrow^j \mathsf{C}^\star, f_i(s), ok$.

That is, in both cases we know there exists j such that $\mathsf{C}; \mathsf{C}^\star, s \rightsquigarrow^j \mathsf{C}^\star, f_i(s), ok$. As such, from (5) and the definition of $\rightsquigarrow^{j+1}$ we have $\mathsf{C}^\star, s \rightsquigarrow^{j+1} \mathsf{C}^\star, f_i(s), ok$, i.e. $\mathsf{C}^\star, s \rightsquigarrow^+ \mathsf{C}^\star, f_i(s), ok$. That is, from (6) we have:

$$\forall i \in \mathbb{N}^+.\ \forall s \in p(i{-}1).\ \mathsf{C}^\star, s \rightsquigarrow^+ \mathsf{C}^\star, f_i(s), ok \land f_i(s) \in p(i) \tag{7}$$

Pick an arbitrary $s \in p(0)$. From (7) we then know $\mathsf{C}^\star, s \rightsquigarrow^+ \mathsf{C}^\star, f_1(s), ok \rightsquigarrow^+ \mathsf{C}^\star, f_2(s), ok \rightsquigarrow^+ \cdots$, as required.

Case Div-Cons

Pick arbitrary p, C such that $\vdash [p]\ \mathsf{C}\ [\infty]$. Pick an arbitrary $s \in p$. From the $p \subseteq p'$ premise we know $s \in p'$. As such, from the $[p']\ \mathsf{C}\ [\infty]$ premise we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $s_1, s_2, \cdots$, such that $\mathsf{C}, s \rightsquigarrow^+ \mathsf{C}_1, s_1, ok \rightsquigarrow^+ \mathsf{C}_2, s_2, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-Subst

Pick arbitrary p, C, q, y such that $y \notin \mathrm{fv}(p, \mathsf{C})$ and $(\vdash [p]\ \mathsf{C}\ [\infty])[y/x]$, i.e. $\vdash [p[y/x]]\ \mathsf{C}[y/x]\ [\infty]$. 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 $\vdash [p]\ \mathsf{C}\ [\infty]$ premise and the inductive hypothesis we then know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $s_1, s_2, \cdots$ such that $\mathsf{C}, s_p \rightsquigarrow^+ \mathsf{C}_1, s_1, ok \rightsquigarrow^+ \mathsf{C}_2, s_2, ok \rightsquigarrow^+ \cdots$. Let $\mathsf{C}'_i = \mathsf{C}_i[y/x]$ and $s'_i = s_i[y \mapsto s'_i(y)]$ for all i. As such, from the semantics we also have $\mathsf{C}[y/x], s \rightsquigarrow^+ \mathsf{C}'_1, s'_1, ok \rightsquigarrow^+ \mathsf{C}'_2, s'_2, ok \rightsquigarrow^+ \cdots$, as required. □

D UNTer Completeness

D.1 Completeness of BUA and FUA Rules

Let us write $\vdash^n_{\mathrm{B}}$ $[p]$ C $[\epsilon : q]$ to denote that $\vdash_{\mathrm{B}}$ $[p]$ C $[\epsilon : q]$ can be derived by n or fewer applications of $\vdash_{\mathrm{B}}$ rules, i.e. $n \in \mathbb{N}^+$ denotes the maximum depth of the derivation tree. Note that, $\vdash_{\mathrm{B}}$ $[p]$ C $[\epsilon : q] \Leftrightarrow \exists n.\ \vdash^n_{\mathrm{B}}$ $[p]$ C $[\epsilon : q]$.

Given a command C, let us write C^0 for skip and C^n for $\underbrace{\mathrm{C}; \cdots ; \mathrm{C}}_{n \text{ times}}$, for all $n > 0$.

Lemma 11. *For all $k \in \mathbb{N}^+$, $n \in \mathbb{N}$, p and q, if $\vdash^k_{\mathrm{B}}$ $[p]$ C^n $[ok\colon q]$ is derivable, then $\vdash_{\mathrm{B}}$ $[p]$ $\mathrm{C}^\star$ $[ok\colon q]$ is also derivable.*

Proof. Pick arbitrary $k \in \mathbb{N}^+$, $n \in \mathbb{N}$, p and q such that $\vdash^k_{\mathrm{B}}$ $[p]$ C^n $[ok\colon q]$ holds. We then proceed by induction on k.

Base case $k = 1$
As $\vdash_{\mathrm{B}}$ $[p]$ C^n $[ok\colon q]$ holds and $k = 1$, by inversion on $\vdash_{\mathrm{B}}$ rules we know that $n = 0$, i.e. $\mathrm{C}^0 = \mathrm{skip}$ and thus $p = q$. Consequently, from Loop0 and since $p = q$ we have $\vdash_{\mathrm{B}}$ $[p]$ $\mathrm{C}^\star$ $[ok\colon q]$, as required.

Case $k = j{+}1$
$\mathrm{C}^n = \mathrm{C}; \mathrm{C}^i$. As $\vdash^k_{\mathrm{B}}$ $[p]$ C^n $[ok\colon q]$ holds and $k > 1$, by inversion on the $\vdash_{\mathrm{B}}$ rules we know that the only applicable rules are Subst or Disj or ConsB or Seq. That is, we have one of the following four derivations for some p', q', $p_{i\in I}$, $q_{i\in I}$ and r:

$$\frac{\vdash^j_{\mathrm{B}}\ [p']\ \mathrm{C}^n\ [ok\colon q'] \qquad x \notin \mathsf{fv}(p', q')}{\vdash^1_{\mathrm{B}}\ [p]\ \mathrm{C}^n\ [ok\colon q]}\ (\textsc{Subst})$$

$$\frac{\vdash^j_{\mathrm{B}}\ [p_i]\ \mathrm{C}^n\ [ok\colon q_i] \qquad \text{for all } i \in I}{\vdash^1_{\mathrm{B}}\ [p]\ \mathrm{C}^n\ [ok\colon q]}\ (\textsc{Disj})$$

$$\frac{\vdash^j_{\mathrm{B}}\ [p]\ \mathrm{C}^n\ [ok\colon q'] \qquad q' \subseteq q}{\vdash^1_{\mathrm{B}}\ [p]\ \mathrm{C}^n\ [ok\colon q]}\ (\textsc{ConsB})$$

$$\frac{\vdash^j_{\mathrm{B}}\ [p']\ \mathrm{C}^n\ [ok\colon q'] \qquad \mathsf{fv}(r) \cap \mathsf{mod}(\mathrm{C}^n) = \emptyset}{\vdash^1_{\mathrm{B}}\ [p]\ \mathrm{C}^n\ [ok\colon q]}\ (\textsc{Constancy})$$

$$\frac{\vdash^j_{\mathrm{B}}\ [p]\ \mathrm{C}^{n-1}\ [ok\colon r] \qquad \vdash^j_{\mathrm{B}}\ [r]\ \mathrm{C}\ [ok\colon q]}{\vdash^1_{\mathrm{B}}\ [p]\ \mathrm{C}^n\ [ok\colon q]}\ (\textsc{Seq})$$

where $p = p'[y/x]$ and $q = q'[y/x]$ in the first derivation; $p = \bigvee_{i\in I} p_i$ and $q = \bigvee_{i \in I} q_i$ in the second derivation; $p = p' \wedge r$ and $q = q' \wedge r$ in the fourth derivation; and $n > 0$ in the last derivation. Note that in the last derivation we are also making use of the fact that $\mathrm{C}^n = \mathrm{C}^{n-1}; \mathrm{C}$ when $n > 0$.

In the case of the first derivation, from the $\vdash^j_{\mathrm{B}}$ $[p']$ C^n $[ok\colon q']$ premise and the inductive hypothesis we have $\vdash_{\mathrm{B}}$ $[p']$ $\mathrm{C}^\star$ $[ok\colon q']$. Consequently, since $x \notin \mathsf{fv}(p', q')$, $p = p'[y/x]$ and $q = q'[y/x]$, from Subst we can derive $\vdash_{\mathrm{B}}$ $[p]$ $\mathrm{C}^\star$ $[ok\colon q]$, as required.

Similarly, in the case of the second derivation, from the premise and the inductive hypothesis we have $\vdash_{\mathrm{B}}$ $[p_i]$ $\mathrm{C}^\star$ $[ok\colon q_i]$ for all $i \in I$. Consequently, since $p = \bigvee_{i\in I} p_i$ and $q = \bigvee_{i\in I} q_i$, from Disj we can derive $\vdash_{\mathrm{B}}$ $[p]$ $\mathrm{C}^\star$ $[ok\colon q]$, as required.

In the case of the third derivation,from the premise and the inductive hypothesis we have $\vdash_{\mathrm{B}}$ $[p]$ $\mathrm{C}^\star$ $[ok\colon q']$. Consequently, since $q' \subseteq q$, from ConsB we can derive $\vdash_{\mathrm{B}}$ $[p]$ $\mathrm{C}^\star$ $[ok\colon q]$, as required.

In the case of the fourth derivation, from the $\vdash^{j}_{\mathrm{B}}\ [p']\ \mathsf{C}^{n}\ [ok\colon q']$ premise and the inductive hypothesis we have $\vdash^{j}_{\mathrm{B}}\ [p']\ \mathsf{C}^{\star}\ [ok\colon q']$. As such, from the $\mathsf{fv}(r) \cap \mathsf{mod}(\mathsf{C}^{n}) = \emptyset$ and since $p = p' \land r$ and $q = q' \land r$, we can apply the Constancy rule and derive $\vdash^{j}_{\mathrm{B}}\ [p]\ \mathsf{C}^{\star}\ [ok\colon q]$, as required.

In the case of the last derivation, from the $\vdash^{j}_{\mathrm{B}}\ [p]\ \mathsf{C}^{n-1}\ [ok\colon r]$ premise and the inductive hypotheses we also have $\vdash_{\mathrm{B}}\ [p]\ \mathsf{C}^{\star}\ [ok\colon r]$. Consequently, from the $\vdash^{j}_{\mathrm{B}}\ [r]\ \mathsf{C}\ [ok\colon q]$ premise and an application of the Loop rule we can derive $\vdash_{\mathrm{B}}\ [p]\ \mathsf{C}^{\star}\ [ok\colon q]$, as required. □

Lemma 12 (BUA completeness). *For all p, C, q, ϵ, if $\models_{\mathrm{B}}\ [p]\ \mathsf{C}\ [\epsilon : q]$ holds, then $\vdash_{\mathrm{B}}\ [p]\ \mathsf{C}\ [\epsilon : q]$ can be proven using the proof rules in Fig. 1.*

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

Case $\mathsf{C} = \mathsf{skip}$

Pick arbitrary p, q such that $\models_{\mathrm{B}}\ [p]\ \mathsf{skip}\ [\epsilon : q]$ holds. Given the semantics of skip, we then know $p \subseteq q$. As such, we can derive $\vdash_{\mathrm{B}}\ [p]\ \mathsf{C}\ [\epsilon : q]$ using Skip and ConsB.

Cases $\mathsf{C} = \mathsf{assume}(B)$ **and** $\mathsf{C} = \mathsf{error}$

The proofs of these cases are analogous to the $\mathsf{C} = \mathsf{skip}$ case and omitted.

Case $\mathsf{C} = x := e$

Pick arbitrary p such that $\models_{\mathrm{B}}\ [p]\ x := e\ [ok\colon q]$ holds. As $\exists y.\ p[y/x] \land x = e[y/x]$ is the strongest post of $x := e$ from p (see [O'Hearn 2019]), we then know $\exists y.\ p[y/x] \land x = e[y/x] \subseteq q$. Moreover, from Assign we have $\vdash_{\mathrm{B}}\ [p]\ x := e\ [ok\colon \exists y.\ p[y/x] \land x = e[y/x]]$. Consequently, as $\exists y.\ p[y/x] \land x = e[y/x] \subseteq q$, from ConsB we have $\vdash_{\mathrm{B}}\ [p]\ x := e\ [ok\colon q]$, as required.

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

Pick arbitrary p, q such that $\models_{\mathrm{B}}\ [p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\epsilon : q]$ holds. From the definition of $\models_{\mathrm{B}}\ [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : q]$ we know p and q can be partitioned as $p = \bigvee_{i \in I}\{s_i\}$, $q = q_1 \lor q_2$ and $q_1 = \bigvee_{i \in I}\{s'_i\}$ such that $\models_{\mathrm{B}}\ [\{s_i\}]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\epsilon : \{s'_i\}]$ for $i \in I$. Pick an arbitrary y such that $y \notin \mathsf{fv}(\mathsf{C})$, $y \notin \mathsf{fv}(p)$ and $y \notin \mathsf{fv}(q)$. Then we know that $\mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]$ is semantically equivalent to $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}$ and thus $\models_{\mathrm{B}}\ [\{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\epsilon : \{s'_i\}]$ also holds for $i \in I$. From the inductive hypothesis we then have$\vdash_{\mathrm{B}}\ [\{s_i\}]\ \mathsf{C}[y/x]\ [\epsilon : \{s'_i\}]$ holds for $i \in I$ and thus from Local we have $\vdash_{\mathrm{B}}\ [\exists y.\ \{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\epsilon : \exists y.\ \{s'_i\}]$ for $i \in I$. From ConsEq we then have $\vdash_{\mathrm{B}}\ [\{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\epsilon : \{s'_i\}]$ for $i \in I$. Moreover, as $p = \bigvee_{i \in I}\{s_i\}$ and $q_1 = \bigvee_{i \in I}\{s'_i\}$, from Disj we have $\vdash_{\mathrm{B}}\ [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\epsilon : q_1]$. As $q = q_1 \lor q_2$ and thus $q_1 \subseteq q$, from ConsB we have $\vdash_{\mathrm{B}}\ [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\epsilon : q]$. On the other hand, since $y \notin \mathsf{fv}(\mathsf{C})$, $y \notin \mathsf{fv}(p)$ and $y \notin \mathsf{fv}(q)$, we know $p[x/y] \Leftrightarrow p$, $q[x/y] \Leftrightarrow q$ and thus from Subst and ConsEq we have $\vdash_{\mathrm{B}}\ [p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\epsilon : q]$, as required.

Case $\mathsf{C} = \mathsf{C}_1; \mathsf{C}_2$

Pick arbitrary p, q such that $\models_{\mathrm{B}}\ [p]\ \mathsf{C}_1; \mathsf{C}_2\ [\epsilon : q]$ holds. From the semantics of $\mathsf{C}_1; \mathsf{C}_2$ we then know either 1) $\epsilon = ok$ and there exists r such that $\models_{\mathrm{B}}\ [p]\ \mathsf{C}_1\ [ok\colon r]$ and $\models_{\mathrm{B}}\ [r]\ \mathsf{C}_2\ [\epsilon : q]$; or 2) there exists p_1, p_2, q_1, q_2, r such that $p = p_1 \lor p_2$, $q = q_1 \lor q_2$, $\models_{\mathrm{B}}\ [p_1]\ \mathsf{C}_1\ [er\colon q_1]$, $\models_{\mathrm{B}}\ [p_2]\ \mathsf{C}_1\ [ok\colon r]$ and $\models_{\mathrm{B}}\ [r]\ \mathsf{C}_2\ [er\colon q_2]$. In case (1) from $\models_{\mathrm{B}}\ [p]\ \mathsf{C}_1\ [ok\colon r]$ and $\models_{\mathrm{B}}\ [r]\ \mathsf{C}_2\ [\epsilon : q]$ and the inductive hypotheses we know we can prove $\vdash_{\mathrm{B}}\ [p]\ \mathsf{C}_1\ [ok\colon r]$ and $\vdash_{\mathrm{B}}\ [r]\ \mathsf{C}_2\ [\epsilon : q]$. Consequently, using Seq we can prove $\vdash_{\mathrm{B}}\ [p]\ \mathsf{C}_1; \mathsf{C}_2\ [\epsilon : q]$, as required.

In case (2) from $\models_B [p_1]\ C_1\ [er\colon q_1]$, $\models_B [p_2]\ C_1\ [ok\colon r]$, $\models_B [r]\ C_2\ [er\colon q_2]$ and the inductive hypotheses we have $\vdash_B [p_1]\ C_1\ [er\colon q_1]$, $\vdash_B [p_2]\ C_1\ [ok\colon r]$ and $\vdash_B [r]\ C_2\ [er\colon q_2]$. As such, from $\vdash_B [p_1]\ C_1\ [er\colon q_1]$ and SeqEr we have $\vdash_B [p_1]\ C_1; C_2\ [er\colon q_1]$. Similarly, from $\vdash_B [p_2]\ C_1\ [ok\colon r]$ and $\vdash_B [r]\ C_2\ [er\colon q_2]$ and Seq we have $\vdash_B [p_2]\ C_1; C_2\ [er\colon q_2]$. Consequently, from $\vdash_B [p_1]\ C_1; C_2\ [er\colon q_1]$, $\vdash_B [p_2]\ C_1; C_2\ [er\colon q_2]$, Disj and since $p = p_1 \lor p_2$ and $q = q_1 \lor q_2$ we have $\vdash_B [p]\ C_1; C_2\ [er\colon q]$, 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 p_1, p_2, q_1, q_2, r such that $p = p_1 \lor p_2$, $q = q_1 \lor q_2$, $\models_B [p_1]\ C_1\ [\epsilon : q_1]$ and $\models_B [p_2]\ C_2\ [\epsilon : q_2]$. From $\models_B [p_1]\ C_1\ [\epsilon : q_1]$, $\models_B [p_2]\ C_2\ [\epsilon : q_2]$ and the inductive hypotheses we know we can prove $\vdash_B [p_1]\ C_1\ [\epsilon : q_1]$, $\vdash_B [p_2]\ C_2\ [\epsilon : q_2]$, and thus using Choice we can prove $\vdash_B [p_1]\ C_1 + C_2\ [\epsilon : q_1]$, $\vdash_B [p_2]\ C_1 + C_2\ [\epsilon : q_2]$. As such, using Disj and since $p = p_1 \lor p_2$, $q = q_1 \lor q_2$, we can prove $\vdash_B [p]\ C_1 + C_2\ [\epsilon : q]$, as required.

Case $C = C^\star$
Pick arbitrary p, q such that $\models_B [p]\ C^\star\ [\epsilon : q]$ holds. There are two cases to consider: 1) $\epsilon = ok$; or 2) $\epsilon = er$. In case (1), we know that each state $s_p \in p$, reaches some state $s_q \in q$ after a number of iterations. That is, we can partition p and q such that $p = \bigvee_{i \in I} p_i$, $q = q' \lor q''$, for some q' and q'' with $q' = \bigvee_{i \in I} q_i$, and for all $i \in I$ we have $\models_B [p_i]\ C^i\ [ok\colon q_i]$. As such, by the inductive hypothesis we have $\vdash_B [p_i]\ C^i\ [ok\colon q_i]$ for all $i \in I$, and thus from Lemma 11 we have $\vdash_B [p_i]\ C^\star\ [ok\colon q_i]$ for all $i \in I$. Using Disj and given that $p = \bigvee_{i \in I} p_i$ and $q' = \bigvee_{i \in I} q_i$, we then have $\vdash_B [p]\ C^\star\ [ok\colon q']$. Finally, as $q = q' \lor q''$ and thus $q' \subseteq q$, we can apply the ConsB and get $\vdash_B [p]\ C^\star\ [ok\colon q]$, as required.

In case (2), from the semantics of loops we know that starting from each $s_p \in p$, C executed normally for a number of (possibly zero) iterations, and in the subsequent iteration the loop encountered an error. That is, we can partition p and q such that $p = \bigvee_{i \in I} p_i$, $q = q' \lor q''$, for some q' and q'' with $q' = \bigvee_{i \in I} q_i$, and for all $i \in I$ we know there exists r_i such that $\models_B [p_i]\ C^i\ [ok\colon r_i]$ and $\models_B [r_i]\ C\ [er\colon q_i]$. From $\models_B [p_i]\ C^i\ [ok\colon r_i]$ and the inductive hypothesis we have $\vdash_B [p_i]\ C^i\ [ok\colon r_i]$, and thus (from Lemma 11) we have $\vdash_B [p_i]\ C^\star\ [ok\colon r_i]$ for all $i \in I$. Consequently, from Disj and since $p = \bigvee_{i \in I} p_i$, we have $\vdash_B [p]\ C^\star\ [ok\colon \bigvee_{i \in I} r_i]$. On the other hand, from $\models_B [r_i]\ C\ [er\colon q_i]$ and the inductive hypothesis we have $\vdash_B [r_i]\ C\ [er\colon q_i]$ for all $i \in I$. As such, from Disj and since $q' = \bigvee_{i \in I} q_i$, we have $\vdash_B [\bigvee_{i \in I} r_i]\ C^\star\ [er\colon q']$. Therefore, as $q = q' \lor q''$ and thus $q' \subseteq q$, we can apply the ConsB and get $\vdash_B [\bigvee_{i \in I} r_i]\ C\ [er\colon q]$, as required. Consequently, from $\vdash_B [p]\ C^\star\ [ok\colon \bigvee_{i \in I} r_i]$, $\vdash_B [\bigvee_{i \in I} r_i]\ C\ [er\colon q]$ and Seq we have $\vdash_B [p]\ C^\star; C\ [er\colon q]$. As such, from Loop we have $\vdash_B [p]\ C^\star\ [er\colon q]$, as required. □

Lemma 13 (FUA completeness). *For all p, C, q, ϵ, if $\models_F [p]\ C\ [\epsilon : q]$ holds, then $\vdash_F [p]\ C\ [\epsilon : q]$ can be proven using the proof rules in Fig. 1.*

Proof. The proof of this lemma is as given by O'Hearn [2019]. □

Theorem 15 (Completeness). *For all p, C, q, ϵ, if $\models_\dagger [p]\ C\ [\epsilon : q]$ holds, then $\vdash_\dagger [p]\ C\ [\epsilon : q]$ can be proven using the proof rules in Fig. 1.*

Proof. Follows immediately from Lemma 12 and Lemma 13. □

D.2 Completeness of Divergence Rules

In what follows, we write $C, s \rightsquigarrow^{+} C', s', \epsilon$ for $\exists n.\ C, s \rightsquigarrow^{n} C', s', \epsilon$.

Theorem 16 (Divergence completeness). *For all p, C, if $\models [p]\ \mathsf{C}\ [\infty]$ holds, then $\vdash [p]\ \mathsf{C}\ [\infty]$ can be proven using the proof rules in Fig. 2,.*

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

Cases $\mathsf{C} = \mathsf{skip}$, $\mathsf{C} = x := e$, $\mathsf{C} = \mathsf{error}$, $\mathsf{C} = \mathsf{assume}(B)$
These cases do not arise as they have no divergent steps and reduce to skip in either 0 or 1 steps.

Case $\mathsf{C} = \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}$
Pick arbitrary p such that $\models [p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\infty]$ holds. From the definition of $\models [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\infty]$ we know p can be partitioned as $p = \bigvee_{i \in I}\{s_i\}$ such that $\models [\{s_i\}]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\infty]$ for $i \in I$. Pick an arbitrary y such that $y \notin \mathsf{fv}(\mathsf{C})$ and $y \notin \mathsf{fv}(p)$. Then we know that $\mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]$ is semantically equivalent to $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}$ and thus $\models [\{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\infty]$ also holds for $i \in I$. From the inductive hypothesis we then have$\vdash [\{s_i\}]\ \mathsf{C}[y/x]\ [\infty]$ can be derived for $i \in I$ and thus from Div-Local we have $\vdash [\exists y.\ \{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\infty]$ for $i \in I$. As $\exists y.\ \{s_i\} \Leftrightarrow \{s_i\}$, from Div-Cons we then have $\vdash [\{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\infty]$ for $i \in I$. Moreover, as $p = \bigvee_{i \in I}\{s_i\}$, from Div-Disj we have $\vdash [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}[y/x]\ [\infty]$. On the other hand, since $y \notin \mathsf{fv}(\mathsf{C})$ and $y \notin \mathsf{fv}(p)$, we know $p[x/y] \Leftrightarrow p$, and thus from Div-Subst and ConsEq we have $\vdash [p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\infty]$, as required.

Pick an arbitrary y such that $y \notin \mathsf{fv}(\mathsf{C})$ and $y \notin \mathsf{fv}(p)$. Then we know that $\mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}$ is semantically equivalent to $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}$ and thus $\models [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\infty]$ holds. From the inductive hypothesis we then have$\vdash [p]\ \mathsf{C}\ [\infty]$, and thus from Div-Local we have $\vdash [\exists y.\ p \land y = v_1]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\infty]$.

From the definition of $\models [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\infty]$ we know p can be partitioned as $p = \bigvee_{i \in I}\{s_i\}$, $q = q_1 \lor q_2$ and $q_1 = \bigvee_{i \in I}\{s'_i\}$ such that $\models_{\mathrm{B}} [\{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : \{s'_i\}]$ for $i \in I$.

From the semantics of $\mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}$ we know there exist v_1 such that $\models [p \land y = v_1]\ \mathsf{C}\ [\infty]$ holds. From the inductive hypothesis we then have$\vdash [p \land y = v_1]\ \mathsf{C}\ [\infty]$, and thus from Div-Local we have $\vdash [\exists y.\ p \land y = v_1]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\infty]$. As $y \notin \mathsf{fv}(p)$, using Div-Cons we have $\vdash [p \land \exists y.\ y = v_1]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\infty]$. Once again, using Div-Cons we have $\vdash [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\infty]$. Finally, using Div-Subst and since $y \notin \mathsf{fv}(\mathsf{C})$ and $y \notin \mathsf{fv}(p)$, we have $\vdash [p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\infty]$, as required.

From the definition of $\models_{\mathrm{B}} [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : q]$ we know p and q can be partitioned as $p = \bigvee_{i \in I}\{s_i\}$, $q = q_1 \lor q_2$ and $q_1 = \bigvee_{i \in I}\{s'_i\}$ such that $\models_{\mathrm{B}} [\{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : \{s'_i\}]$ for $i \in I$. Moreover, from the semantics of $\mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}$ we know there exist v_i, v'_i for $i \in I$ such that $\models_{\mathrm{B}} [\{s_i\} \land y = v_i]\ \mathsf{C}\ [\epsilon : \{s'_i\} \land y = v'_i]$ holds for $i \in I$. From the inductive hypothesis we then have$\vdash_{\mathrm{B}} [\{s_i\} \land y = v_i]\ \mathsf{C}\ [\epsilon : \{s'_i\} \land y = v'_i]$ holds for $i \in I$ and thus from Local we have $\vdash_{\mathrm{B}} [\exists y.\ \{s_i\} \land y = v_i]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : \exists y.\ \{s'_i\} \land y = v'_i]$ for $i \in I$. From ConsEq we then have $\vdash_{\mathrm{B}} [\{s_i\}]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : \{s'_i\}]$ for $i \in I$. Moreover, as $p = \bigvee_{i \in I}\{s_i\}$ and $q_1 = \bigvee_{i \in I}\{s'_i\}$, from Disj we have $\vdash_{\mathrm{B}} [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : q_1]$. Finally, as $q = q_1 \lor q_2$ and thus $q_1 \subseteq q$, from ConsB we have $\vdash_{\mathrm{B}} [p]\ \mathsf{local}\ y\ \mathsf{in}\ \mathsf{C}\ [\epsilon : q]$, as required.

Case $\mathsf{C} = \mathsf{C}_1; \mathsf{C}_2$
Pick arbitrary p such that $\models [p]\ \mathsf{C}_1; \mathsf{C}_2\ [\infty]$ holds. From the semantics of $\mathsf{C}_1; \mathsf{C}_2$ we then know there exist p_1, p_2, q such that $p = p_1 \lor p_2$, $\models [p_1]\ \mathsf{C}_1\ [\infty]$, $\models_{\mathrm{B}} [p_2]\ \mathsf{C}_1\ [ok\colon q]$ and $\models [q]\ \mathsf{C}_2\ [\infty]$. From $\models [p_1]\ \mathsf{C}_1\ [\infty]$, $\models [q]\ \mathsf{C}_2\ [\infty]$ and the inductive hypotheses we have $\vdash [p_1]\ \mathsf{C}_1\ [\infty]$ and $\vdash [q]\ \mathsf{C}_2\ [\infty]$. Similarly, from $\models_{\mathrm{B}} [p_2]\ \mathsf{C}_1\ [ok\colon q]$ and Theorem 15 we have $\vdash_{\mathrm{B}} [p_2]\ \mathsf{C}_1\ [ok\colon q]$. As such, from $\vdash [p_1]\ \mathsf{C}_1\ [\infty]$ and Div-Seq1 we have $\vdash [p_1]\ \mathsf{C}_1; \mathsf{C}_2\ [\infty]$. Similarly, from $\vdash_{\mathrm{B}} [p_2]\ \mathsf{C}_1\ [ok\colon q]$

and $\vdash [q]\ C_2\ [\infty]$ and Div-Seq2 we have $\vdash [p_2]\ C_1; C_2\ [\infty]$. Consequently, from $\vdash [p_1]\ C_1; C_2\ [\infty]$, $\vdash [p_2]\ C_1; C_2\ [\infty]$, Div-Disj and since $p = p_1 \vee p_2$ we have $\vdash [p]\ C_1; C_2\ [\infty]$, 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 exist p_1, p_2 such that $p = p_1 \vee p_2$, $\models [p_1]\ C_1\ [\infty]$ and $\models [p_2]\ C_2\ [\infty]$. From $\models [p_1]\ C_1\ [\infty]$, $\models [p_2]\ C_2\ [\infty]$ and the inductive hypothesis we know we can prove $\vdash [p_1]\ C_1\ [\infty]$, and $\vdash [p_2]\ C_2\ [\infty]$. As such, using Div-Choice we can prove $\vdash [p_1]\ C_1 + C_2\ [\infty]$ and $\vdash [p_2]\ C_1 + C_2\ [\infty]$. Consequently, using Div-Disj and since $p = p_1 \vee p_2$ we can prove $\models [p]\ C_1 + C_2\ [\infty]$, as required.

Case $C = C^\star$
Pick arbitrary p such that $\models [p]\ C^\star\ [\infty]$ 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) $\models_B [p(n)]\ C\ [ok\colon p(n{+}1)]$ for all $n \in \mathbb{N}$; or
2) there exists q_i for $i \in I$ such that p can be partitioned as $p = \bigvee_{i \in I} p_i$ with $\models_B [p_i]\ C^i\ [ok\colon q_i]$ and $\models [q_i]\ C\ [\infty]$ for $i \in I$.

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

In case (2), from $\models_B [p_i]\ C^i\ [ok\colon q_i]$ and Theorem 15 we have $\vdash_B [p_i]\ C^i\ [ok\colon q_i]$ and thus from Lemma 11 for each $i \in I$ we have:

$$\vdash_B [p_i]\ C^i\ [ok\colon q_i] \tag{8}$$

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

$$\vdash [q_i]\ C\ [\infty] \tag{9}$$

As $C^i = \underbrace{C; \cdots; C}_{i \text{ times}}$, we can then prove $\vdash [p_i]\ C^\star\ [\infty]$ as follows for each $i \in I$:

$$\dfrac{\dfrac{\dfrac{(8)}{\vdash_B [p_i]\ C^i\ [ok\colon q_i]} \qquad \dfrac{\dfrac{(9)}{\vdash [q_i]\ C\ [\infty]}}{\vdash [q_i]\ C^\star\ [\infty]}\ \text{Div-LoopNest}}{\vdash [p_i]\ C^i; C^\star\ [\infty]}\ \text{Div-Seq2}}{\vdash [p_i]\ C^\star\ [\infty]}\ \text{Div-LoopUnfold} \times i$$

Consequently, as $p = \bigvee_{i \in I} p_i$, using Div-Disj we have $\vdash [p]\ C^\star\ [\infty]$, as required. □

E The Relation Between FUA and BUA Triples

Theorem 17. *For all* p, C, q, ϵ, *if* $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ *holds and* $\mathsf{min}_{\mathsf{pre}}(p, \mathsf{C}, q)$*, then* $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ *also holds, where*

$$\mathsf{min}_{\mathsf{pre}}(p, \mathsf{C}, q) \overset{def}{\iff} \forall p'.\ p' \subset p \Rightarrow \not\models_{\mathsf{F}} [p']\ \mathsf{C}\ [\epsilon : q]$$

Proof. Pick arbitrary p, C, q, ϵ such that $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds and $\mathsf{min}_{\mathsf{pre}}(p, \mathsf{C}, q)$. Let us proceed by contradiction and assume that $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ does not hold. That is, there exists $s_p \in p$ such that:

$$\neg\exists s_q \in q, n.\ \mathsf{C}, s_p \xrightarrow{n} -, s_q, \epsilon \tag{10}$$

Let $p' \triangleq p \setminus \{s_p\}$. We next show that $\models_{\mathsf{F}} [p']\ \mathsf{C}\ [\epsilon : q]$ holds.
Pick an arbitrary $s_2 \in q$. Since $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds, from its definition we know there exists $s_1 \in p$, k such that $\mathsf{C}, s_1 \xrightarrow{k} -, s_2, \epsilon$. However, from (10) we know $s_1 \neq s_p$. Consequently, since $p' \triangleq p \setminus \{s_p\}$ and $s_1 \in p$, we know $s_1 \in p'$. That is, there exists $s_1 \in p'$, k such that $\mathsf{C}, s_1 \xrightarrow{k} -, s_2, \epsilon$, and thus by definition we have:

$$\models_{\mathsf{F}} [p']\ \mathsf{C}\ [\epsilon : q] \tag{11}$$

Finally, from $\mathsf{min}_{\mathsf{pre}}(p, \mathsf{C}, q)$, (11) and the definition of $\mathsf{min}_{\mathsf{pre}}$ we have $p' \not\subset p$. This, however, leads to a contradiction as $p' \triangleq p \setminus \{s_p\}$ and thus $p' \subset p$. □

Theorem 18. *For all* p, C, q, ϵ, *if* $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ *holds and* $\mathsf{min}_{\mathsf{post}}(p, \mathsf{C}, q)$*, then* $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ *also holds, where*

$$\mathsf{min}_{\mathsf{post}}(p, \mathsf{C}, q) \overset{def}{\iff} \forall q'.\ q' \subset q \Rightarrow \not\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q']$$

Proof. Pick arbitrary p, C, q, ϵ such that $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds and $\mathsf{min}_{\mathsf{post}}(p, \mathsf{C}, q)$. Let us proceed by contradiction and assume that $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ does not hold. That is, there exists $s_q \in q$ such that:

$$\neg\exists s_p \in p, n.\ \mathsf{C}, s_p \xrightarrow{n} -, s_q, \epsilon \tag{12}$$

Let $q' \triangleq q \setminus \{s_q\}$. We next show that $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q']$ holds.
Pick an arbitrary $s_1 \in p$. Since $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds, from its definition we know there exists $s_2 \in q$, k such that $\mathsf{C}, s_1 \xrightarrow{k} -, s_2, \epsilon$. However, from (12) we know $s_2 \neq s_q$. Consequently, since $q' \triangleq q \setminus \{s_q\}$ and $s_2 \in q$, we know $s_q \in q'$. That is, there exists $s_2 \in q'$, k such that $\mathsf{C}, s_1 \xrightarrow{k} -, s_2, \epsilon$, and thus by definition we have:

$$\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q'] \tag{13}$$

Finally, from $\mathsf{min}_{\mathsf{post}}(p, \mathsf{C}, q)$, (13) and the definition of $\mathsf{min}_{\mathsf{post}}$ we have $q' \not\subset q$. This, however, leads to a contradiction as $q' \triangleq q \setminus \{s_q\}$ and thus $q' \subset q$. □

F UNTer$^{\text{SL}}$ 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 $\mathsf{mod}(\mathrm{C}) \cap \mathsf{fv}(r) = \emptyset$, and thus this frame remains unchanged when executing C. The $*$ 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).

SL-Frame

$$\frac{\vdash_\dagger [p]\, \mathrm{C}\, [\epsilon : q] \quad \mathsf{mod}(\mathrm{C}) \cap \mathsf{fv}(r) = \emptyset}{\vdash_\dagger [p * r]\ \mathrm{C}\ [\epsilon : q * r]}$$

SL-Free

$$\vdash_{\mathsf{F}} [x \mapsto v]\, \mathsf{free}(x)\, [ok \colon \mathsf{emp}]$$

ISL-Free

$$\vdash_{\mathsf{F}} [x \mapsto v]\, \mathsf{free}(x)\, [ok \colon x \not\mapsto]$$

The compositionality afforded by SL-Frame allows us to devise *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 SL-Frame. For instance, we can describe the behaviour of freeing memory as in the 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': $x \mapsto v * x \mapsto v' \Leftrightarrow \mathsf{false}$. On the other hand, the emp assertion describes states in which the heap is empty, and thus represents the identity of $*$-composition; i.e. for all p: $p * \mathsf{emp} \Leftrightarrow p$.

FUA Triples and Separation Logic. To achieve compositional reasoning, an SL extension of a FUA reasoning system must preserve the soundness of SL-Frame. However, as Raad et al. [2020] show, the original model of SL is unsound for FUA reasoning. in Particular, we can apply SL-Frame with $r \triangleq x \mapsto v$ as shown below, resulting in an invalid FUA triple:

$$\cfrac{\cfrac{\cfrac{}{\vdash_{\mathsf{F}}\ [x \mapsto v]\ \mathsf{free}(x)\ [ok\colon \mathsf{emp}]}\ \text{SL-Free}}{\nvdash_{\mathsf{F}}\ [x \mapsto v * x \mapsto v]\ \mathsf{free}(x)\ [ok\colon \mathsf{emp} * x \mapsto v]}\ \text{SL-Frame}}{\nvdash_{\mathsf{F}}\ [\mathsf{false}]\ \mathsf{free}(x)\ [ok\colon x \mapsto v]}\ (\text{semantics of } *)$$

Note that $[\mathsf{false}]\ \mathsf{free}(x)\ [ok\colon x \mapsto v]$ 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. [2020] 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 $*$-conjunction. More concretely, one cannot consistently $*$-conjoin $x \not\mapsto$ either with $x \mapsto -$ or with itself: $x \mapsto - * x \not\mapsto \ \Leftrightarrow \mathsf{false}$ and $x \not\mapsto * x \not\mapsto \ \Leftrightarrow \mathsf{false}$. Using negative assertions, one can specify the $\mathsf{free}(x)$ axiom as in ISL-Free above (right), recovering the frame rule: this time, when we frame $x \mapsto v$ on both sides, we obtain the inconsistent assertion $x \mapsto - * x \not\mapsto$ on the right-hand side (i.e. we have false as both pre- and post-states), which always renders a FUA triple vacuously valid.

UNTer$^{\text{SL}}$ Proof Rules. We present UNTer$^{\text{SL}}$ proof rules in Fig. 10. Note that all UNTer rules (both BUA and FUA) in Fig. 1, except Constancy and Assign, are also UNTer$^{\text{SL}}$ rules and are omitted from Fig. 10. In particular, we replace Constancy with the more powerful Frame rule and give a

SkipSL
$\vdash_{\dagger} [\mathsf{emp}]\ \mathsf{skip}\ [ok\colon \mathsf{emp}]$

AssumeSL
$\vdash_{\dagger} [B]\ \mathsf{assume}(B)\ [ok\colon B]$

AssignSL
$\vdash_{\dagger} [x{=}x']\ x:=e\ [ok\colon x{=}e[x'/x]]$

Alloc
$\vdash_{\dagger} [\mathsf{emp}]\, x:=\mathsf{alloc}()\, [ok\colon \exists l.\ l\mapsto v * x{=}l]$

AllocFree
$\vdash_{\dagger} [y \not\mapsto]\, x:=\mathsf{alloc}()\, [ok\colon y\mapsto v * x{=}y]$

Free
$\vdash_{\dagger} [x\mapsto e]\ \mathsf{free}(x)\ [ok\colon x \not\mapsto]$

FreeEr
$\vdash_{\dagger} [x \not\mapsto]\ \mathsf{free}(x)\ [er\colon x \not\mapsto]$

FreeNull
$\vdash_{\dagger} [x{=}\mathsf{null}]\ \mathsf{free}(x)\ [er\colon x{=}\mathsf{null}]$

Store
$\vdash_{\dagger} [x\mapsto e]\ [x]:=y\ [ok\colon x\mapsto y]$

StoreEr
$\vdash_{\dagger} [x \not\mapsto]\ [x]:=y\ [er\colon x \not\mapsto]$

StoreNull
$\vdash_{\dagger} [x{=}\mathsf{null}]\ [x]:=y\ [er\colon x{=}\mathsf{null}]$

Load
$\vdash_{\dagger} [x{=}x' * y\mapsto e]\ x:=[y]\ [ok\colon x{=}e[x'/x] * y\mapsto e[x'/x]]$

LoadEr
$\vdash_{\dagger} [y \not\mapsto]\ x:=[y]\ [er\colon y \not\mapsto]$

LoadNull
$\vdash_{\dagger} [y{=}\mathsf{null}]\ x:=[y]\ [er\colon y{=}\mathsf{null}]$

Frame
$$\frac{\vdash_{\dagger} [p]\,\mathsf{C}\,[\epsilon : q] \qquad \mathsf{mod}(\mathsf{C}) \cap \mathsf{fv}(r) = \emptyset}{\vdash_{\dagger} [p * r]\ \mathsf{C}\ [\epsilon : q * r]}$$

Div-Frame
$$\frac{\vdash\ [p]\ \mathsf{C}\ [\infty]}{\vdash\ [p * r]\ \mathsf{C}\ [\infty]}$$

Fig. 10. UNTer$^{\mathrm{SL}}$ proof rules where x and x' are distinct variables and † in each rule can be instantiated as F or B; all UNTer rules in Fig. 1 (except Assign, Constancy) and Fig. 2 are also valid in UNTer$^{\mathrm{SL}}$ and are omitted.

local rule for assignment (see below). The BUA rules in Fig. 10 are identical to those of ISL. As with ISL (and in contrast to UNTer), UNTer$^{\mathrm{SL}}$ 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. This is particularly useful when the expression e on the right-hand side of the assignment refers to x (e.g. $x := x{+}1$). Similarly, assume(B) requires no resource and only stipulates that B hold in the set of pre-states which is unchanged in the post-states.

As in SL and ISL, the crux of UNTer$^{\mathrm{SL}}$ lies in the Frame rule, allowing one to extend the pre- and post-states with disjoint resources in r, where fv(r) returns the set of free variables in r, and mod(C) returns the set of (program) variables modified by C (i.e. those on the left-hand of ':=' in assignment, lookup and allocation). These definitions are standard and elided. Note that the Skip and Assume of UNTer can be derived from their UNTer$^{\mathrm{SL}}$ counterparts using Frame.

The rules for heap-manipulating operations are identical to those of ISL. In particular, Alloc rule allows us to allocate a previously unallocated location, while Load, Store and Free describe the successful execution of heap lookup, mutation and deallocation, respectively. As in ISL, negative assertions allow us to detect memory safety violations when accessing deallocated locations. For instance, FreeEr states that attempting to deallocate x causes an error when x is already deallocated; *mutatis mutandis* for LoadEr and StoreEr. As shown in AllocFree, we can use negative assertions to allocate a previously deallocated location: if y is deallocated ($y \not\mapsto$ holds in the pre-state), then it may be reallocated. Finally, the FreeNull, LoadNull and StoreNull rules state that accessing x causes an error when x is null.

UNTer$^{\mathrm{SL}}$ Model and Assertion Semantics. As well as a (variable) store, in UNTer$^{\mathrm{SL}}$ each state additionally includes a *heap* (memory); i.e. an UNTer$^{\mathrm{SL}}$ state, $\sigma \in \textsc{State}^{\mathrm{SL}} \triangleq \textsc{Store} \times \textsc{Heap}$, is a pair of the form (s, h), comprising a store $s \in \textsc{Store} \triangleq \textsc{Var} \to \textsc{Val}$ (as in UNTer) and a *heap* $h \in \textsc{Heap}$. We present the semantics of UNTer$^{\mathrm{SL}}$ assertions at the top of Fig. 11, where an assertion is interpreted as a set of UNTer$^{\text{SL}}$ states. The semantics of classical assertions (imported from UNTer) are standard and omitted; e.g. the semantics of $e \oplus e'$ is given as the set of pairs of the form $(s, \emptyset)$ such that $s(e) \oplus s(e')$ holds, where $\emptyset$ is the empty heap (with empty domain).

Small-Step Operational Semantics. As with UNTer, we define the UNTer$^{\text{SL}}$ semantics through small-step transitions. Specifically, as seen in SL-Local–SL-Loop, the UNTer$^{\text{SL}}$ semantics of constructs imported from UNTer are analogous to their UNTer counterparts and are simply lifted to operate on UNTer$^{\text{SL}}$ 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 deallocating 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.

Semantic BUA, FUA and Divergent triples in UNTer^{SL}. The formal interpretations of BUA, FUA and divergent triples in UNTer$^{\text{SL}}$ are identical to their UNTer counterparts, except that the UNTer states (stores) are replaced with corresponding UNTer$^{\text{SL}}$ states (pairs of stores and heaps).

More concretely, a BUA triple in UNTer$^{\text{SL}}$ is *valid*, written $\models_{\mathsf{B}} [p]\ \mathrm{C}\ [\epsilon : q]$, iff for all $\sigma_p \in p$, there exists $\sigma_q \in q$ and n such that $\mathrm{C}, \sigma_p \xrightarrow{n} -, \sigma_q, \epsilon$, where $\mathrm{C}, \sigma \xrightarrow{n} -, \sigma', \epsilon$ is as defined in Def. 1 with the UNTer states s, s', s'' replaced with corresponding UNTer$^{\text{SL}}$ states σ, σ' and σ'' as follows:

$$\begin{aligned}
\mathrm{C}, \sigma \xrightarrow{n} \mathrm{C}', \sigma', \epsilon \ \overset{\text{def}}{\iff}\ & (n{=}0 \wedge \mathrm{C}{=}\mathrm{C}'{=}\text{skip} \wedge \sigma{=}\sigma' \wedge \epsilon{=}ok) \\
& \vee (n{=}1 \wedge \epsilon \in \text{ErExit} \wedge \exists s, h, s'.\ \mathrm{C}, \sigma \rightarrow \mathrm{C}', (s, h), \epsilon \wedge \mathrm{C}', s \leadsto_{\text{er}}^{+} \text{skip}, s' \wedge \sigma' = (s', h)) \\
& \vee (\exists k, \mathrm{C}'', \sigma''.\ n{=}k{+}1 \wedge \mathrm{C}, \sigma \rightarrow \mathrm{C}'', \sigma'', ok \wedge \mathrm{C}'', \sigma'' \xrightarrow{k} \mathrm{C}', \sigma', \epsilon)
\end{aligned}$$

and where $\mathrm{C}, \sigma \rightarrow \mathrm{C}', \sigma', \epsilon$ corresponds to UNTer$^{\text{SL}}$ transitions in Fig. 11, and $\leadsto_{\text{er}}^{+}$ denotes the transitive closure of the $\leadsto_{\text{er}}$ transitions in Fig. 6.

A FUA triple in UNTer$^{\text{SL}}$ is *valid*, written $\models_{\mathsf{F}} [p]\ \mathrm{C}\ [\epsilon : q]$, iff for all $\sigma_q \in q$, there exists $\sigma_p \in p$ and n such that $\mathrm{C}, \sigma_p \xrightarrow{n} -, \sigma_q, \epsilon$. Analogously, a divergent triple in UNTer$^{\text{SL}}$ is *valid*, written $\models [p]\ \mathrm{C}\ [\infty]$, iff for all $\sigma \in p$, there exists an infinite series of $\mathbb{C}_1, \mathbb{C}_2, \cdots, \sigma_1, \sigma_2, \cdots$ and $n_1, n_2, \cdots$ such that $\mathrm{C}, \sigma \rightsquigarrow^{n_1} \mathbb{C}_1, \sigma_1, ok \rightsquigarrow^{n_2} \mathbb{C}_2, \sigma_2, ok \rightsquigarrow^{n_3} \cdots$, where where the chain $\mathrm{C}, \sigma \rightsquigarrow^{n_1} \mathbb{C}_1, \sigma_1, ok \rightsquigarrow^{n_2} \mathbb{C}_2, \sigma_2, ok \rightsquigarrow^{n_3} \cdots$ is a shorthand for $\mathrm{C}, \sigma \rightsquigarrow^{n_1} \mathbb{C}_1, \sigma_1, ok \wedge \mathbb{C}_1, \sigma_1 \rightsquigarrow^{n_2} \mathbb{C}_2, \sigma_2, ok \wedge \cdots$, $\rightsquigarrow^{n}$ and $\rightsquigarrow^{n}$ is as defined below:

$$\begin{aligned}
\mathbb{C}, \sigma \rightsquigarrow^{n} \mathbb{C}', \sigma', \epsilon \ \overset{\text{def}}{\iff}\ & (n = 1 \wedge \epsilon = ok \wedge \mathbb{C}, \sigma \rightarrow \mathbb{C}', \sigma', \epsilon) \\
& \vee (n{=}1 \wedge \epsilon \in \text{ErExit} \wedge \exists s, h, s'.\ \mathrm{C}, \sigma \rightarrow \mathrm{C}', (s, h), \epsilon \wedge \mathrm{C}', s \leadsto_{\text{er}}^{+} \text{skip}, s' \wedge \sigma' = (s', h)) \\
& \vee (\exists k, \sigma'', \mathbb{C}''.\ n{=}k{+}1 \wedge \mathbb{C}, \sigma \rightarrow \mathbb{C}'', \sigma'', ok \wedge \mathbb{C}'', \sigma'' \rightsquigarrow^{k} \mathbb{C}', \sigma', \epsilon)
\end{aligned}$$

Finally, in §G we show that the BUA, FUA and divergent proof system of UNTer$^{\text{SL}}$ presented in Fig. 10 is sound. That is, for all p, q, C and ϵ:

1) if $\vdash_{\mathsf{B}} [p]\ \mathrm{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10, then $\models_{\mathsf{B}} [p]\ \mathrm{C}\ [\epsilon : q]$ holds;
2) if $\vdash_{\mathsf{F}} [p]\ \mathrm{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10, then $\models_{\mathsf{F}} [p]\ \mathrm{C}\ [\epsilon : q]$ holds; and
3) if $\vdash [p]\ \mathrm{C}\ [\infty]$ is derivable using the rules in Fig. 10, then $\models [p]\ \mathrm{C}\ [\infty]$ holds.

$$
\begin{aligned}
(\!|.|\!) &: \textsc{Ast} \rightarrow \mathcal{P}(\textsc{State}^{\text{SL}}) \\
(\!|\text{emp}|\!) &\triangleq \left\{(s, h) \;\middle|\; \mathit{dom}(h)=\emptyset\right\} \\
(\!|e \mapsto e'|\!) &\triangleq \left\{(s, h) \;\middle|\; \mathit{dom}(h)=\{s(e)\} \land h(s(e))=s(e')\neq\bot\right\} \\
(\!|e \not\mapsto\, |\!) &\triangleq \left\{(s, h) \;\middle|\; \mathit{dom}(h)=\{s(e)\} \land h(s(e))=\bot\right\} \\
(\!|p * q|\!) &\triangleq \left\{\sigma_p \circ \sigma_q \;\middle|\; \sigma_p \in (\!|p|\!) \land \sigma_q \in (\!|q|\!)\right\}
\end{aligned}
$$

$$
\text{where} \qquad (s, h) \circ (s', h') \triangleq \begin{cases} (s, h \uplus h') & \text{if } s=s' \land \mathit{dom}(h_1) \cap \mathit{dom}(h_2)=\emptyset \land \text{wf}(h \uplus h') \\ \text{undefined} & \text{otherwise} \end{cases}
$$

$$
\textsc{SL-Local}\ \frac{s' = s[x \mapsto v] \qquad v \in \textsc{Val}}{\text{local } x \text{ in C}, (s, h) \rightarrow \text{C}; \text{end}(x, s(x)), (s', h), \mathit{ok}}
\qquad
\textsc{SL-LocalEnd}\ \frac{s' = s[x \mapsto v]}{\text{end}(x, v), (s, h) \rightarrow \text{skip}, (s', h), \mathit{ok}}
\qquad
\textsc{SL-Assign}\ \frac{s' = s[x \mapsto s(e)]}{x := e, (s, h) \rightarrow \text{skip}, (s', h), \mathit{ok}}
$$

$$
\textsc{SL-Assume}\ \frac{\sigma = (s, -) \qquad s(B) = \text{true}}{\text{assume}(B), \sigma \rightarrow \text{skip}, \sigma, \mathit{ok}}
\qquad
\textsc{SL-Error}\ \ \text{error}, \sigma \rightarrow \text{skip}, \sigma, \mathit{er}
\qquad
\textsc{SL-Choice}\ \frac{i \in \{1, 2\}}{\text{C}_1 + \text{C}_2, \sigma \rightarrow \text{C}_i, \sigma, \mathit{ok}}
$$

$$
\textsc{SL-Seq1}\ \frac{\mathbb{C}_1, \sigma \rightarrow \mathbb{C}_1', \sigma', \epsilon}{\mathbb{C}_1; \mathbb{C}_2, \sigma \rightarrow \mathbb{C}_1'; \mathbb{C}_2, \sigma', \epsilon}
\qquad
\textsc{SL-SeqSkip}\ \ \text{skip}; \mathbb{C}, \sigma \rightarrow \mathbb{C}, \sigma, \mathit{ok}
\qquad
\textsc{SL-Loop0}\ \ \text{C}^\star, \sigma \rightarrow \text{skip}, \sigma, \mathit{ok}
\qquad
\textsc{SL-Loop}\ \ \text{C}^\star, \sigma \rightarrow \text{C}; \text{C}^\star, \sigma, \mathit{ok}
$$

$$
\textsc{SL-Alloc}\ \frac{l \notin \mathit{dom}(h) \qquad h' = h \uplus [l \mapsto v] \qquad s' = s[x \mapsto l]}{x := \text{alloc}(), (s, h) \rightarrow \text{skip}, (s', h'), \mathit{ok}}
\qquad
\textsc{SL-AllocFree}\ \frac{h(l) = \bot \quad h' = h[l \mapsto v] \quad s' = s[x \mapsto l]}{x := \text{alloc}(), (s, h) \rightarrow \text{skip}, (s', h'), \mathit{ok}}
$$

$$
\textsc{SL-Free}\ \frac{s(x) = l \quad h(l) \in \textsc{Val} \quad h' = h[l \mapsto \bot]}{\text{free}(x), (s, h) \rightarrow \text{skip}, (s, h'), \mathit{ok}}
\qquad
\textsc{SL-FreeEr}\ \frac{s(x) = \text{null} \lor h(s(x)) = \bot}{\text{free}(x), (s, h) \rightarrow \text{skip}, (s, h), \mathit{er}}
\qquad
\textsc{SL-Load}\ \frac{h(s(y)) = v \in \textsc{Val} \quad s' = s[x \mapsto v]}{x := [y], (s, h) \rightarrow \text{skip}, (s', h), \mathit{ok}}
$$

$$
\textsc{SL-LoadEr}\ \frac{s(y) = \text{null} \lor h(s(y)) = \bot}{x := [y], (s, h) \rightarrow \text{skip}, (s, h), \mathit{er}}
\qquad
\textsc{SL-Store}\ \frac{s(y) = l \quad h(l) \in \textsc{Val} \quad h' = h[l \mapsto s(y)]}{[x] := y, (s, h) \rightarrow \text{skip}, (s, h'), \mathit{ok}}
\qquad
\textsc{SL-StoreEr}\ \frac{s(x) = \text{null} \lor h(s(x)) = \bot}{[x] := y, (s, h) \rightarrow \text{skip}, (s, h), \mathit{er}}
$$

Fig. 11. The semantics of UNTER$^{\text{SL}}$ assertions (above); the UNTER$^{\text{SL}}$ small-step operational semantics (below)

G UNTER$^{\text{SL}}$ Soundness

Definition 3.

$$s_1 \sim_A s_2 \overset{\text{def}}{\iff} \forall x \in A.\ s_1(x){=}s_2(x)$$

Definition 4.

$$h_p \ \#\ h \overset{\text{def}}{\iff} dom(h_p) \cap dom(h){=}\emptyset$$
$$\sigma_p \ \#\ \sigma \overset{\text{def}}{\iff} \exists \sigma'.\ \sigma_p \circ \sigma = \sigma'$$

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

Proposition 4. *For all assertions p and all s, s', h, if $(s, h) \in p$ and $s \sim_{\mathsf{fv}(p)} s'$, then $(s', h) \in p$. For all ϵ, C, x, v, n, (s_1, h_1) and (s_2, h_2), if $\mathrm{C}, (s_1, h_1) \xrightarrow{n} -, (s_2, h_2), \epsilon$ and $x \notin \mathsf{fv}(\mathrm{C})$, then $\mathrm{C}, ((s_1[x \mapsto v], h_1) \xrightarrow{n} -, (s_2[x \mapsto v], h_2)), \epsilon$.*

Proposition 5. *For all $n, \mathrm{C}, \sigma, \sigma', \epsilon$, if $\mathrm{C}^\star; \mathrm{C}, \sigma \xrightarrow{n} -, \sigma', \epsilon$ then there exists m such that $\mathrm{C}; \mathrm{C}^\star, \sigma \xrightarrow{m} -, \sigma', \epsilon$.*

Lemma 14. *For all n, σ, σ', $\mathbb{C}$, $\mathbb{C}'$, if $\mathbb{C}, \sigma \xrightarrow{n} \mathbb{C}', \sigma', ok$, then $\mathbb{C}' = \mathsf{skip}$.*

PROOF. The proof of this lemma is analogous to that of Lemma 1 and is omitted here.

Lemma 15. *For all σ, σ', σ'', $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}'$, i, j, ϵ, if $\mathbb{C}_1, \sigma \xrightarrow{i} -, \sigma'', ok$ and $\mathbb{C}_2, \sigma'' \xrightarrow{j} \mathbb{C}', \sigma', \epsilon$, then there exists n such that $\mathbb{C}_1; \mathbb{C}_2, \sigma \xrightarrow{n} \mathbb{C}', \sigma', \epsilon$.*

PROOF. The proof of this lemma is analogous to that of Lemma 2 and is omitted here.

Lemma 16. *For all s, h, s', h', s'', $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}'$, i, if $\mathbb{C}_1, (s, h) \xrightarrow{i} \mathbb{C}', (s', h'), er$, and $\mathbb{C}_2, s' \rightsquigarrow^{+}_{\mathrm{er}} \mathsf{skip}, s''$, then $\mathbb{C}_1; \mathbb{C}_2, (s, h) \xrightarrow{i} \mathbb{C}'; \mathbb{C}_2, (s'', h'), er$.*

PROOF. The proof of this lemma is analogous to that of Lemma 3 and is omitted here.

G.1 BUA Soundness in UNTER$^{\text{SL}}$

Lemma 17. *For all p, C, a, b,*
if

$$\forall n \in \mathbb{N}, s, h_p, h.\ a \leq n < b \land (s, h_p) \in p(n) \land h_p \ \#\ h \Rightarrow$$
$$\exists (s', h_q) {\in} p(n{+}1), j.\ s \sim_{\overline{\mathsf{mod}(\mathrm{C})}} s' \land \mathrm{C}, (s, h_p \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), ok$$

then

$$\forall k, i \in \mathbb{N}, s, h_p, h.\ a \leq i \land i{+}k < b \land (s, h_p) \in p(i) \land h_p \ \#\ h \Rightarrow$$
$$\exists (s', h_q) \in p(i{+}k), j.\ s \sim_{\overline{\mathsf{mod}(\mathrm{C}^\star)}} s' \land \mathrm{C}^\star, (s, h_p \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), ok$$

PROOF. Pick arbitrary p, C, a, b such that:

$$\forall n \in \mathbb{N}, s, h_p, h.\ a \leq n < b \land (s, h_p) \in p(n) \land h_p \ \#\ h \Rightarrow$$
$$\exists (s', h_q) {\in} p(n{+}1), j.\ s \sim_{\overline{\mathsf{mod}(\mathrm{C})}} s' \land \mathrm{C}, (s, h_p \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), ok \tag{14}$$

Pick an arbitrary k. We proceed by induction on k.

Base case $k{=}0$
Pick an arbitrary $i \in \mathbb{N}$, s, h_p, h such that $a \leq i \land i{+}k < b$, $(s, h_p) \in p(i)$ and $\land h_p \ \#\ h$. We

then simply have $s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star)}} s$. From S-Loop0 we have $\mathsf{C}^\star, (s, h_p \uplus h) \rightarrow \mathsf{skip}, (s, h_p \uplus h), ok$. As such, as we have $\mathsf{skip}, (s, h_p \uplus h) \xrightarrow{0} \mathsf{skip}, (s, h_p \uplus h), ok$ (from the definition of $\xrightarrow{0}$), by definition we have $\mathsf{C}^\star, (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_p \uplus h), ok$. Consequently, we have $(s, h_p) \in p(i)$ and $\mathsf{C}^\star, (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_p \uplus h), ok$, as required.

Inductive case k=t+1

$$\begin{array}{l}\forall c \in \mathbb{N}, s, h_p, h.\ a \leq c \land c{+}t < b \land (s, h_p) \in p(c) \land h_p \# h \Rightarrow \\ \qquad \exists (s', h_q) \in p(c{+}t), j.\ s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star)}} s' \land \mathsf{C}^\star, (s, h_p \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), ok\end{array} \tag{I.H}$$

Pick an arbitrary $i \in \mathbb{N}$, $(s, h_p) \in p(i)$ and h such that $h_p \# h$. As $a \leq i \land i{+}k < b$, $(s, h_p) \in p(i)$ and $h_p \# h$, from (14) we know we know there exists $(s_i, h_i) \in p(i{+}1)$ and m such that $s \sim_{\overline{\mathsf{mod}(\mathsf{C})}} s_i$ and $\mathsf{C}, (s, h_p \uplus h) \xrightarrow{m} -, (s_i, h_i \uplus h), ok$. That is, $h_i \# h$. As $s \sim_{\overline{\mathsf{mod}(\mathsf{C})}} s_i$ and $\mathsf{mod}(\mathsf{C}) = \mathsf{mod}(\mathsf{C}^\star)$, we also have $s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star)}} s_i$.

On the other hand, since $(s_i, h_i) \in p(i{+}1)$ and $h_i \# h$, from (I.H) we know there exists $(s', h_q) \in p(i{+}1{+}t)$ and b such that $s_i \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star)}} s' \land \mathsf{C}^\star, (s_i, h_i \uplus h) \xrightarrow{b} -, (s', h_q \uplus h), ok$. That is, $(s', h_q) \in p(i{+}k)$. Therefore, from Lemma 15, $\mathsf{C}, (s, h_p \uplus h) \xrightarrow{m} -, (s_i, h_i \uplus h), ok$ and $\mathsf{C}^\star, (s_i, h_i \uplus h), \xrightarrow{b} -, (s', h_q \uplus h), ok$ we know there exists c such that $\mathsf{C}; \mathsf{C}^\star, (s, h_p \uplus h), \xrightarrow{c} -, (s', h_q \uplus h), ok$.

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

Lemma 18. *For all* $p, \mathsf{C}, q, \epsilon$, *if* $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ *can be derived using the proof rules in Fig. 10, then:*

$$\begin{array}{l}\forall (s_p, h_p) \in p.\ \forall h.\ h_p \# h \implies \\ \qquad \exists (s_q, h_q) \in q, n.\ s_p \sim_{\overline{\mathsf{mod}(\mathsf{C})}} s_q \land \mathsf{C}, (s_p, h_p \uplus h) \xrightarrow{n} -, (s_q, h_q \uplus h), \epsilon\end{array}$$

Proof. By induction on the structure of rules in Fig. 10.

Case Skip

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

Case AssignSL

Pick an arbitrary $\sigma_p \in x = x'$ and an arbitrary h such that $h_p \# h$. 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 $\mathsf{mod}(x := e) = \{x\}$, by definition of s' we have $s \sim_{\overline{\mathsf{mod}(x := e)}} s'$. From SL-Assign we then have $x := e, (s, h) \rightarrow \mathsf{skip}, (s', h), ok$. As such, since we also have $\mathsf{skip}, (s', h) \xrightarrow{0} \mathsf{skip}, (s', h), ok$, by definition we have $x := e, (s, h) \xrightarrow{1} \mathsf{skip}, (s', h), ok$, i.e. $x := e, (s, \emptyset \uplus h) \xrightarrow{1} \mathsf{skip}, (s', \emptyset \uplus h), ok$

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 $(s', \emptyset) \in x = e[x'/x]$, $s \sim_{\overline{\mathsf{mod}(x := e)}} s'$ and $x := e, (s, \emptyset \uplus h) \xrightarrow{1} \mathsf{skip}, (s', \emptyset \uplus h), ok$, as required.

Case Assume

Pick arbitrary p, B such that $\vdash_{\mathrm{B}} \big[p \wedge B\big]\ \mathsf{assume}(B)\ \big[ok\colon p \wedge B\big]$. Pick an arbitrary $(s, h_p) \in p \wedge B$ and an arbitrary h such that $h_p \,\#\, h$. By definition we then know $s(B) = \mathsf{true}$. As $\mathsf{mod}(\mathsf{assume}(B)) = \emptyset$, by definition we have $s \sim_{\overline{\mathsf{mod}(\mathsf{assume}(B))}} s$. From S-Assume we then have $\mathsf{assume}(B), (s, h_p \uplus h) \longrightarrow \mathsf{skip}, (s, h_p \uplus h), ok$. As such, since we also have $\mathsf{skip}, (s, h_p \uplus h) \xrightarrow{0} \mathsf{skip}, (s, h_p \uplus h), ok$, by definition we have $\mathsf{assume}(B), (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_p \uplus h), ok$. Consequently, we have $(s, h_p) \in p \wedge B$, $s \sim_{\overline{\mathsf{mod}(\mathsf{assume}(B))}} s$ and $\mathsf{assume}(B), (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_p \uplus h), ok$, as required.

Case AssumeSL

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

Case Error

Pick arbitrary p such that $\vdash_{\mathrm{B}} [p]\ \mathsf{error}\ [er\colon p]$. Pick an arbitrary $(s, h_p) \in p$ and an arbitrary h such that $h_p \,\#\, h$. Let $\sigma = (s, h_p \uplus h)$. From S-Error we then have $\mathsf{error}, \sigma \longrightarrow \mathsf{skip}, \sigma, er$. As such, since $(s, h_p) \in p$, by definition we have $\mathsf{error}, \sigma \xrightarrow{1} \mathsf{skip}, \sigma, er$, as required. Moreover, as $\mathsf{mod}(\mathsf{error}) = \emptyset$ we also have $s \sim_{\overline{\mathsf{mod}(\mathsf{error})}} s$, as required.

Case Seq

Pick arbitrary $p, q, r, \mathrm{C}_1, \mathrm{C}_2, \epsilon$ such that $\vdash_{\mathrm{B}} \big[p\big]\ \mathrm{C}_1\ \big[ok\colon r\big]$ and $\vdash_{\mathrm{B}} [r]\ \mathrm{C}_2\ [\epsilon : q]$. Pick an arbitrary $(s, h_p) \in p$ and an arbitrary h such that $h_p \,\#\, h$. From $\vdash_{\mathrm{B}} \big[p\big]\ \mathrm{C}_1\ \big[ok\colon r\big]$ and the inductive hypothesis we then know there exists $(s_r, h_r) \in r, i$ such that $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_1)}} s_r$ and $\mathrm{C}_1, (s, h_p \uplus h) \xrightarrow{i} -, (s_r, h_r \uplus h), ok$. Moreover, as $(s_r, h_r) \in r, i$, from $\vdash_{\mathrm{B}} [r]\ \mathrm{C}_2\ [\epsilon : q]$ and the inductive hypothesis we know there exists $(s', h_q) \in q, j$ such that $s_r \sim_{\overline{\mathsf{mod}(\mathrm{C}_2)}} s'$ and $\mathrm{C}_2, (s_r, h_r \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), \epsilon$. As $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_1)}} s_r$ and $s_r \sim_{\overline{\mathsf{mod}(\mathrm{C}_2)}} s'$, by definition we also have $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_1;\mathrm{C}_2)}} s_r$ and $s_r \sim_{\overline{\mathsf{mod}(\mathrm{C}_1;\mathrm{C}_2)}} s'$, and thus we also have $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_1;\mathrm{C}_2)}} s'$. On the other hand, as $\mathrm{C}_1, (s, h_p \uplus h) \xrightarrow{i} -, (s_r, h_r \uplus h), ok$ and $\mathrm{C}_2, (s_r, h_r \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), \epsilon$, from Lemma 15 we know there exists n such that $\mathrm{C}_1; \mathrm{C}_2, (s, h_p \uplus h) \xrightarrow{n} -, (s', h_q \uplus h), \epsilon$. That is, there exists $(s', h_q) \in q, n$ such that $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_1;\mathrm{C}_2)}} s'$, $\mathrm{C}_1; \mathrm{C}_2, (s, h_p \uplus h) \xrightarrow{n} -, (s', h_q \uplus h), \epsilon$, as required.

Case SeqEr

Pick arbitrary $p, q, \mathrm{C}_1, \mathrm{C}_2$ such that $\vdash_{\mathrm{B}} [p]\ \mathrm{C}_1; \mathrm{C}_2\ [er\colon q]$. Pick an arbitrary $(s, h_p) \in p$ and an arbitrary h such that $h_p \,\#\, h$. From the $\vdash_{\mathrm{B}} [p]\ \mathrm{C}_1\ [er\colon q]$ premise and the inductive hypothesis we then know there exists $(s', h_q) \in q, i$ such that $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_1)}} s'$ and $\mathrm{C}_1, (s, h_p \uplus h) \xrightarrow{i} -, (s', h_q \uplus h), er$. From Prop. 2 we know $\mathrm{C}_2, s' \rightsquigarrow^{+}_{\mathrm{er}} s'$, and thus from $\mathrm{C}_1, s \xrightarrow{i} -, s', er$ and Lemma 16 we know $\mathrm{C}_1; \mathrm{C}_2, (s, h_p \uplus h) \xrightarrow{i} -, (s', h_q \uplus h), er$, as required.

Case Choice

Pick arbitrary $p, q, \mathrm{C}_1, \mathrm{C}_2, \epsilon$ and $i \in \{1, 2\}$ such that $\vdash_{\mathrm{B}} [p]\ \mathrm{C}_1 + \mathrm{C}_2\ [\epsilon : q]$. Pick an arbitrary $(s, h_p) \in p$ and an arbitrary h such that $h_p \,\#\, h$. From the $\vdash_{\mathrm{B}} [p]\ \mathrm{C}_i\ [\epsilon : q]$ premise and the inductive hypothesis we then know there exists $(s', h_q) \in q, i$ such that $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_i)}} s'$ and $\mathrm{C}_i, (s, h_p \uplus h) \xrightarrow{i} -, (s', h_q \uplus h), \epsilon$. As $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_i)}} s'$, by definition we also have $s \sim_{\overline{\mathsf{mod}(\mathrm{C}_1+\mathrm{C}_2)}} s'$ Moreover, from S-Choice we have $\mathrm{C}_1 + \mathrm{C}_2, (s, h_p \uplus h) \longrightarrow \mathrm{C}_i, (s, h_p \uplus h), ok$. As such, from the definition of $\xrightarrow{i+1}$ we have $\mathsf{C}_1 + \mathsf{C}_2, (s, h_p \uplus h) \xrightarrow{i+1} -, (s', h_q \uplus h)', \epsilon$, as required.

Case Loop0

Pick arbitrary p, C such that $\vdash_{\mathsf{B}} [p]\ \mathsf{C}^\star\ [ok\colon p]$. Pick an arbitrary $(s, h_p) \in p$ and an arbitrary h such that $h_p \ \#\ h$. From S-Loop0 we have $\mathsf{C}^\star, (s, h_p \uplus h) \to \mathsf{skip}, (s, h_p \uplus h), ok$. As such, as we have $\mathsf{skip}, (s, h_p \uplus h) \xrightarrow{0} \mathsf{skip}, (s, h_p \uplus h), ok$ (from the definition of $\xrightarrow{0}$), by definition we have $\mathsf{C}^\star, (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_p \uplus h), ok$. Moreover, by definition we have $s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star)}} s$, as required.

Case Loop

Pick arbitrary p, C, q such that $\vdash_{\mathsf{B}} [p]\ \mathsf{C}^\star\ [\epsilon : q]$. Pick an arbitrary $(s, h_p) \in p$ and an arbitrary h such that $h_p \ \#\ h$. From the $\vdash_{\mathsf{B}} [p]\ \mathsf{C}^\star; \mathsf{C}\ [\epsilon : q]$ premise and the inductive hypothesis we know there exists $(s', h_q) \in q, j$ such that $s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star;\mathsf{C})}} s'$ and $\mathsf{C}^\star; \mathsf{C}, (s, h_p \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), \epsilon$. As $s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star;\mathsf{C})}} s'$, by definition we also have $s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star)}} s'$. On the other hand, from Prop. 5 we then know there exists i such that $\mathsf{C}; \mathsf{C}^\star, (s, h_p \uplus h) \xrightarrow{i} -, (s', h_q \uplus h), \epsilon$. From S-Loop we have $\mathsf{C}^\star, (s, h_p \uplus h) \to \mathsf{C}; \mathsf{C}^\star, (s', h_q \uplus h), ok$. As such, from the definition of $\xrightarrow{i+1}$ we have $\mathsf{C}^\star, (s, h_p \uplus h) \xrightarrow{i+1} -, (s', h_q \uplus h), \epsilon$, as required.

Case Loop-Subvar

Pick p, C, k such that $\vdash_{\mathsf{B}} \big[p(0)\big]\ \mathsf{C}^\star\ \big[ok\colon p(k)\big]$. Pick arbitrary $(s, h_p) \in p(0)$ and h such that $h_p \ \#\ h$. From the $\forall n \in \mathbb{N}.\ \vdash_{\mathsf{B}} \big[p(n)\big]\ \mathsf{C}\ \big[ok\colon p(n{+}1)\big]$ premise and the inductive hypothesis we know:

$$\forall n \in \mathbb{N}, (s, h_p) \in p(n), h.\ h_p \ \#\ h \Rightarrow \exists (s', h_q) \in p(n{+}1), j.\ s \sim_{\overline{\mathsf{mod}(\mathsf{C})}} s' \land \mathsf{C}, (s, h_p \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), ok$$

Consequently, from Lemma 17 we know there exists $(s', h_q) \in p(k)$ and j such that $s \sim_{\overline{\mathsf{mod}(\mathsf{C}^\star)}} s'$ and $\mathsf{C}^\star, (s, h_p \uplus h) \xrightarrow{j} -, (s', h_q \uplus h), ok$, as required.

Case Local

Pick arbitrary $p, \mathsf{C}, q, \epsilon$ such that $\vdash_{\mathsf{B}} [\exists x.\ p]\ \mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}\ [\epsilon : \exists x.\ q]$. Pick an arbitrary $(s, h_p) \in \exists x.\ p$ and an arbitrary h such that $h_p \ \#\ h$; i.e. there exists v, s_p such that $s_p = s[x \mapsto v]$ and $(s_p, h_p) \in p$. From the $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ premise and the inductive hypothesis we know there exists $(s_q, h_q) \in q$ and n such that $s_p \sim_{\overline{\mathsf{mod}(\mathsf{C})}} s_q$ and $\mathsf{C}, (s_p, h_p \uplus h) \xrightarrow{n} -, (s_q, h_q \uplus h), \epsilon$. From S-Local we have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, (s, h_p \uplus h) \to \mathsf{C}; \mathsf{end}(x, s(x)), (s_p, h_p \uplus h)$. There are now two cases to consider: 1) $\epsilon{=}ok$; or 2) $\epsilon{=}er$.

In case (1), let $s'' = s_q[x \mapsto s(x)]$. Consequently, as $s_p \sim_{\overline{\mathsf{mod}(\mathsf{C})}} s_q$ and $s''(x) = s(x)$, from the definitions of s_p and s'' we also have $s \sim_{\overline{\mathsf{mod}(\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C})}} s''$. From S-LocalEnd we then have $\mathsf{end}(x, s(x)), (s_q, h_q \uplus h) \to \mathsf{skip}, (s'', h_q \uplus h)$. From the definition of $\xrightarrow{0}$ we have $\mathsf{skip}, (s'', h_q \uplus h) \xrightarrow{0} \mathsf{skip}, (s'', h_q \uplus h), ok$, and thus since we have $\mathsf{end}(x, s(x)), (s_q, h_q \uplus h) \to \mathsf{skip}, (s'', h_q \uplus h)$, from the definition of $\xrightarrow{1}$ we have $\mathsf{end}(x, s(x)), (s_q, h_q \uplus h) \xrightarrow{1} \mathsf{skip}, (s'', h_q \uplus h)$. Consequently, since we also have $\mathsf{C}, (s_p, h_p \uplus h) \xrightarrow{n} -, (s_q, h_q \uplus h), \epsilon$, from Lemma 15 we know there exists m such that $\mathsf{C}; \mathsf{end}(x, s(x)), (s_p, h_p \uplus h) \xrightarrow{m} \mathsf{skip}, (s'', h_q \uplus h), ok$. On the other hand, since we have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, (s, h_p \uplus h) \to \mathsf{C}; \mathsf{end}(x, s(x)), (s_p, h_p \uplus h)$, by definition of $\xrightarrow{m+1}$ we also have $\mathsf{local}\ x\ \mathsf{in}\ \mathsf{C}, (s, h_p \uplus h) \xrightarrow{m+1} \mathsf{skip}, (s'', h_q \uplus h), ok$. Finally, as $(s_q, h_q) \in q$ 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), let $s'' = s_q[x \mapsto s(x)]$. From $\rightsquigarrow_{\text{er}}$ transitions we then have $\text{end}(x, s(x)), s_q \rightsquigarrow_{\text{er}} \text{skip}, s''$. As such, from $\mathsf{C}, (s_p, h_p \uplus h) \xrightarrow{n} -, (s_q, h_q \uplus h), \epsilon$ and Lemma 16 we have $\mathsf{C}; \text{end}(x, s(x)), (s_p, h_p \uplus h) \xrightarrow{n} -, (s'', h_q \uplus h), \epsilon$. On the other hand, since we have $\text{local}\ x\ \text{in}\ \mathsf{C}, (s, h_p \uplus h) \to \mathsf{C}; \text{end}(x, s(x)), (s_p, h_p \uplus h)$, by definition of $\xrightarrow{n+1}$ we also have $\text{local}\ x\ \text{in}\ \mathsf{C}, (s, h_p \uplus h) \xrightarrow{n+1} -, (s'', h_q \uplus h), \epsilon$. Moreover, as $s_p = s[x \mapsto v]$, $\text{mod}(\text{local}\ x\ \text{in}\ \mathsf{C}) = \text{mod}(\mathsf{C}) \cup \{x\}$ and $s_p \sim_{\overline{\text{mod}(\mathsf{C})}} s_q$, by definition we also have $s \sim_{\overline{\text{mod}(\text{local}\ x\ \text{in}\ \mathsf{C})}} s_q$. Finally, as $(s_q, h_q) \in q$ and $s'' = s_q[x \mapsto s(x)]$, by definition we also have $(s'', h_q) \in \exists x.\ q$, as required.

Case Disj

Pick arbitrary $p_1, p_2, q_1, q_2, \mathsf{C}$ such that $\vdash_{\mathsf{B}} [p_1 \vee p_2]\ \mathsf{C}\ [\epsilon : q_1 \vee q_2]$. Pick an arbitrary $(s, h_p) \in p_1 \vee p_2$ and an arbitrary h such that $h_p \ \#\ h$. 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 $\vdash_{\mathsf{B}} [p_1]\ \mathsf{C}\ [\epsilon : q_1]$ premise and the inductive hypothesis we know there exists $(s', h_q) \in q_1, n$ such that $s \sim_{\overline{\text{mod}(\mathsf{C})}} s'$, $\mathsf{C}, (s, h_p \uplus h) \xrightarrow{n} -, (s', h_q \uplus h), \epsilon$. That is, there exists $(s', h_q) \in q_1 \vee q_2$ and n such that $s \sim_{\overline{\text{mod}(\mathsf{C})}} s'$, $\mathsf{C}, (s, h_p \uplus h) \xrightarrow{n} -, (s', h_q \uplus h), \epsilon$, as required. The proof of case (2) is analogous and omitted.

Case DisjTrack

Pick arbitrary $P_1, P_2, Q_1, Q_2, \mathsf{C}$ such that $\vdash_{\mathsf{B}} [P_1 \uplus P_2]\ \mathsf{C}\ [\epsilon : Q_1 \uplus Q_2]$. Pick an arbitrary $i \in dom(P_1 \uplus P_2)$, $(s, h_p) \in (P_1 \uplus P_2)(i)$ and an arbitrary h such that $h_p \ \#\ h$. We then know that either $i \in dom(P_1)$ or $i \in dom(P_2)$. Without loss of generality, let us assume $i \in dom(P_1)$.

As $(s, h_p) \in (P_1 \uplus P_2)(i)$ and $i \in dom(P_1)$, we then have $(s, h_p) \in P_1(i)$. From the $\vdash_{\mathsf{B}} [P_1]\ \mathsf{C}\ [\epsilon : Q_1]$ 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 $s \sim_{\overline{\text{mod}(\mathsf{C})}} s'$ and $\mathsf{C}, (s, h_p \uplus h) \xrightarrow{n} -, (s', h_q \uplus h), \epsilon$. That is, there exists $(s', h_q) \in (Q_1 \uplus Q_2)(i)$ and n such that $s \sim_{\overline{\text{mod}(\mathsf{C})}} s'$ and $\mathsf{C}, (s, h_p \uplus h) \xrightarrow{n} -, (s', h_q \uplus h), \epsilon$, as required.

Case Cons

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

Case Alloc

Pick arbitrary x, v and $(s, h_p) \in p$ and h such that $h_p \ \#\ h$. We then know $h_p \triangleq \emptyset$. Pick l such that $l \notin dom(h)$ and let $h_q = [l \mapsto v]$ and $s' = s[x \mapsto l]$; as such, we also have $(s', h_q) \in l \mapsto v * x = l$ and $s \sim_{\overline{\text{mod}(x := \text{alloc}())}} s'$. Since $l \notin dom(h)$ and $h_q = [l \mapsto v]$, by definition of $\#$ we also know $h_q \ \#\ h$. From SL-Alloc we then have $x := \text{alloc}(), (s, h_p \uplus h) \to \text{skip}, (s', h_q \uplus h), ok$, and since we also have $\text{skip}, (s', h_q \uplus h) \xrightarrow{0} \text{skip}, (s', h_q \uplus h), ok$, by definition of $\xrightarrow{1}$ we have $x := \text{alloc}(), (s, h_p \uplus h) \xrightarrow{1} \text{skip}, (s', h_q \uplus h), ok$, as required.

Case AllocFree

Pick arbitrary x, y and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know there exists l such that $s(y){=}l$ and $h_p \triangleq [l \mapsto \bot]$. Let $h_q{=}[l \mapsto v]$ and $s' = s[x \mapsto l]$; as such, we also have $(s', h_q) \in y \mapsto v * x = y$ and $s \sim_{\overline{\mathsf{mod}(x := \mathsf{alloc}())}} s'$. Since $h_p \ \# \ h$ and $dom(h_p){=}dom(h_q)$, by definition of $\#$ we also know $h_q \ \# \ h$. From SL-AllocFree we then have $x := \mathsf{alloc}(), (s, h_p {\uplus} h) \to \mathsf{skip}, (s', h_q {\uplus} h), ok$, and since we also have $\mathsf{skip}, (s', h_q \uplus h) \xrightarrow{0} \mathsf{skip}, (s', h_q \uplus h), ok$, by definition of $\xrightarrow{1}$ we have $x := \mathsf{alloc}(), (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s', h_q \uplus h), ok$, as required.

Case Free

Pick an arbitrary x and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know there exists l, v such that $s(x){=}l$, $s(e) = v$ and $h_p \triangleq [l \mapsto v]$. Let $h_q{=}[l \mapsto \bot]$; we then have $(s, h_q) \in x \not\mapsto$ and $s \sim_{\overline{\mathsf{mod}(\mathsf{free}(x))}} s$. Since $h_p \ \# \ h$ and $dom(h_p){=}dom(h_q)$, from the definition of $\uplus$ we also know that $h_q \ \# \ h$. From SL-Free we then have $\mathsf{free}(x), (s, h_p \uplus h) \to \mathsf{skip}, (s, h_q \uplus h), ok$, and since we also have $\mathsf{skip}, (s, h_q \uplus h) \xrightarrow{0} \mathsf{skip}, (s, h_q \uplus h), ok$, by definition of $\xrightarrow{1}$ we have $\mathsf{free}(x), (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_q \uplus h), ok$, as required.

Case FreeEr

Pick an arbitrary x and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know there exists l such that $s(x){=}l$ and $h_p \triangleq [l \mapsto \bot]$. Let $h_q{=}h_p$; we then have $(s, h_q) \in x \not\mapsto$ and $s \sim_{\overline{\mathsf{mod}(\mathsf{free}(x))}} s$. From SL-FreeEr we then have $\mathsf{free}(x), (s, h_p \uplus h) \to \mathsf{skip}, (s, h_q \uplus h), er$, and thus by definition of $\xrightarrow{1}$ we have $\mathsf{free}(x), (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_q \uplus h), er$, as required.

Case FreeNull

Pick an arbitrary x and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know $s(x){=}\mathsf{null}$ and $h_p \triangleq \emptyset$. Let $h_q{=}h_p$; we then have $(s, h_q) \in x = \mathsf{null}$ and $s \sim_{\overline{\mathsf{mod}(\mathsf{free}(x))}} s$. From SL-FreeEr we then have $\mathsf{free}(x), (s, h_p \uplus h) \to \mathsf{skip}, (s, h_q \uplus h), er$, and thus by definition of $\xrightarrow{1}$ we have $\mathsf{free}(x), (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_q \uplus h), er$, as required

Case Store

Pick an arbitrary x and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know there exists l, v, v_y such that $s(x){=}l$, $s(y) = v_y$, $s(e) = v$ and $h_p \triangleq [l \mapsto v]$. Let $h_q{=}[l \mapsto v_y]$; we then have $(s, h_q) \in x {\mapsto} y$ and $s \sim_{\overline{\mathsf{mod}([x] := y)}} s$. Since $h_p \ \# \ h$ and $dom(h_p){=}dom(h_q)$, from the definition of $\uplus$ we also know that $h_q \ \# \ h$. From SL-Store we then have $[x] := y, (s, h_p \uplus h) \to \mathsf{skip}, (s, h_q \uplus h), ok$, and since we also have $\mathsf{skip}, (s, h_q \uplus h) \xrightarrow{0} \mathsf{skip}, (s, h_q \uplus h), ok$, by definition of $\xrightarrow{1}$ we have $[x] := y, (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_q \uplus h), ok$, as required.

Case StoreEr

Pick an arbitrary x and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know there exists l such that $s(x){=}l$ and $h_p \triangleq [l \mapsto \bot]$. Let $h_q{=}h_p$; we then have $(s, h_q) \in x \not\mapsto$ and $s \sim_{\overline{\mathsf{mod}([x] := y)}} s$. From SL-StoreEr we then have $[x] := y, (s, h_p \uplus h) \to \mathsf{skip}, (s, h_q \uplus h), er$ and thus by definition of $\xrightarrow{1}$ we have $[x] := y, (s, h_p \uplus h) \xrightarrow{1} \mathsf{skip}, (s, h_q \uplus h), er$, as required.

Case StoreNull
Pick an arbitrary x and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know $s(x)$=null and $h_p \triangleq \emptyset$. Let h_q=h_p; we then have $(s, h_q) \in x = \text{null}$ and $s \sim_{\overline{\text{mod}([x] := y)}} s$. From SL-StoreEr we then have $[x] := y, (s, h_p \uplus h) \longrightarrow \text{skip}, (s, h_q \uplus h), er$, and thus by definition of $\xrightarrow{1}$ we have $[x] := y, (s, h_p \uplus h) \xrightarrow{1} \text{skip}, (s, h_q \uplus h), er$, as required.

Case Load
Pick arbitrary x and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know there exists l, v, v_x such that $s(x) = s(x') = v_x, s(y)=l, s(e) = v$ and $h_p \triangleq [l \mapsto v]$. Let h_q=h_p and $s' = s[x \mapsto v]$; as such, we also have $(s', h_q) \in x = e[x'/x] * y \mapsto e[x'/x]$ and $s \sim_{\overline{\text{mod}(x := [y])}} s'$. Since $h_p \ \# \ h$ and h_p=h_q, we also know $h_q \ \# \ h$. From SL-Load we then have $x := [y], (s, h_p \uplus h) \longrightarrow \text{skip}, (s', h_q \uplus h), ok$, and since we also have $\text{skip}, (s', h_q \uplus h) \xrightarrow{0} \text{skip}, (s', h_q \uplus h), ok$, by definition of $\xrightarrow{1}$ we have $x := [y], (s, h_p \uplus h) \xrightarrow{1} \text{skip}, (s', h_q \uplus h), ok$, as required.

Case LoadEr
Pick an arbitrary y and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know there exists l such that $s(y)$=l and $h_p \triangleq [l \mapsto \bot]$. Let h_q=h_p; we then have $(s, h_q) \in y \not\mapsto$ and $s \sim_{\overline{\text{mod}(x := [y])}} s$. From SL-LoadEr we then have $x := [y], (s, h_p \uplus h) \longrightarrow \text{skip}, (s, h_q \uplus h), er$ and thus by definition of $\xrightarrow{1}$ we have $x := [y], (s, h_p \uplus h) \xrightarrow{1} \text{skip}, (s, h_q \uplus h), er$, as required.

Case LoadNull
Pick an arbitrary y and $(s, h_p) \in p$ and h such that $h_p \ \# \ h$. We then know $s(y)$=null and $h_p \triangleq \emptyset$. Let h_q=h_p; we then have $(s, h_q) \in y = \text{null}$ and $s \sim_{\overline{\text{mod}(x := [y])}} s$. From SL-LoadEr we then have $x := [y], (s, h_p \uplus h) \longrightarrow \text{skip}, (s, h_q \uplus h), er$, and thus by definition of $\xrightarrow{1}$ we have $x := [y], (s, h_p \uplus h) \xrightarrow{1} \text{skip}, (s, h_q \uplus h), er$, as required.

Case Frame
Pick arbitrary $(s_1, h_1) \in p * r$ and h such that $h_1 \ \# \ h$. 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 $h_1 \triangleq h_p \uplus h_r$. From the definition of # and $\uplus$ we then also have $h_p \ \# \ h_r \uplus h$. On the other hand, from the premise of Frame we have $\vdash_B [p]\ \text{C}\ [\epsilon : q]$ and thus from the inductive hypothesis we know there exists s_2, h_q, n such that $s_1 \sim_{\overline{\text{mod}(\text{C})}} s_2$, $(s_2, h_q) \in q$ and $\text{C}, (s_1, h_p \uplus h_r \uplus h) \xrightarrow{n} -, (s_2, h_q \uplus h_r \uplus h), \epsilon$. Moreover, since $s_1 \sim_{\overline{\text{mod}(\text{C})}} s_2$ and as from the premise of Frame we have $\text{mod}(\text{C}) \cap \text{fv}(r)$=$\emptyset$, we also have $s_1 \sim_{\text{fv}(r)} s_2$. Consequently, since $(s_1, h_r) \in r$, from Prop. 4 we have $(s_2, h_r) \in r$. As such from the definition of $*$ we have $(s_2, h_q \uplus h_r) \in q * r$. That is, we know there exists s_2 and h_2=$h_q \uplus h_r$ such that $s_1 \sim_{\overline{\text{mod}(\text{C})}} s_2$, $(s_2, h_2) \in q * r$ and $\text{C}, (s_1, h_1 \uplus h) \xrightarrow{n} -, (s_2, h_2 \uplus h), \epsilon$, as required. □

Lemma 19 (BUA soundness in UNTer$^{\text{SL}}$). *For all p, C, q, ϵ, if $\vdash_B [p]\ \text{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10, then $\models_B [p]\ \text{C}\ [\epsilon : q]$ holds.*

Proof. Pick arbitrary p, C, q, ϵ such that $\vdash_B [p]\ \text{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10. 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 $\text{C}, (s_p, h_p) \xrightarrow{n} -, (s_q, h_q), \epsilon$.

Let $h_0 = \emptyset$ denote the empty heap (with an empty domain). From the definition of $\uplus$ and $\#$ we then know that $h_p \ \# \ h_0$. As such, from Lemma 18 we know there exists $(s_q, h_q) \in q$ and n such that $\mathsf{C}, (s_p, h_p \uplus h_0) \xrightarrow{n} -, (s_q, h_q \uplus h_0), \epsilon$. That is, there exists $(s_q, h_q) \in q$ and n such that $\mathsf{C}, (s_p, h_p) \xrightarrow{n} -, (s_q, h_q), \epsilon$, as required. □

G.2 FUA Soundness in UNTer$^{\text{SL}}$

Lemma 20. *For all $p, \mathsf{C}, q, \epsilon$, if $\vdash_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ can be derived using the proof rules in Fig. 10, then:*

$$\begin{aligned}&\forall (s_q, h_q) \in q.\ \forall h.\ h_q \ \# \ h \implies \\ &\qquad \exists (s_p, h_p) \in p, n.\ s_p \sim_{\overline{\mathsf{mod}(\mathsf{C})}} s_q \wedge \mathsf{C}, (s_p, h_p \uplus h) \xrightarrow{n} -, (s_q, h_q \uplus h), \epsilon\end{aligned}$$

Proof. The proof of this lemma is analogous to that of Lemma 18 and is omitted. □

Lemma 21 (FUA soundness in UNTer$^{\text{SL}}$). *For all $p, \mathsf{C}, q, \epsilon$, if $\vdash_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10, then $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds.*

Proof. Pick arbitrary $p, \mathsf{C}, q, \epsilon$ such that $\vdash_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10. 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 $\mathsf{C}, (s_p, h_p) \xrightarrow{n} -, (s_q, h_q), \epsilon$.

Let $h_0 = \emptyset$ denote the empty heap (with an empty domain). From the definition of $\uplus$ and $\#$ we then know that $h_q \ \# \ h_0$. As such, from Lemma 20 we know there exists $(s_p, h_p) \in p$ and n such that $\mathsf{C}, (s_p, h_p \uplus h_0) \xrightarrow{n} -, (s_q, h_q \uplus h_0), \epsilon$. That is, there exists $(s_p, h_p) \in p$ and n such that $\mathsf{C}, (s_p, h_p) \xrightarrow{n} -, (s_q, h_q), \epsilon$, as required. □

G.3 Divergent Soundness in UNTer$^{\text{SL}}$

Lemma 22. *For all $\mathbb{C}, \sigma, \mathbb{C}', \sigma', \epsilon, n$, if $n > 0$ and $\mathbb{C}, \sigma \xrightarrow{n} \mathbb{C}', \sigma', \epsilon$, then $\mathbb{C}, \sigma \rightsquigarrow^n \mathbb{C}', \sigma', \epsilon$.*

Proof. The proof of this lemma is analogous to that of Lemma 7 and is omitted. □

Lemma 23. *For all $n, \mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_1', \sigma, \mathbb{C}', \sigma'$, if $\mathbb{C}_1, \sigma \rightsquigarrow^n \mathbb{C}_1', \sigma', ok$, then $\mathbb{C}_1; \mathbb{C}_2, \sigma \rightsquigarrow^n \mathbb{C}_1'; \mathbb{C}_2, \sigma', ok$.*

Proof. The proof of this lemma is analogous to that of Lemma 8 and is omitted. □

Lemma 24. *For all $\sigma, \sigma', \sigma'', \mathbb{C}_1, \mathbb{C}_2, \mathbb{C}', i, j, \epsilon$, if $\mathbb{C}_1, \sigma \xrightarrow{i} -, \sigma'', ok$ and $\mathbb{C}_2, \sigma'' \rightsquigarrow^j \mathbb{C}', \sigma', \epsilon$, then there exists n such that $\mathbb{C}_1; \mathbb{C}_2, \sigma \rightsquigarrow^n \mathbb{C}', \sigma', \epsilon$.*

Proof. The proof of this lemma is analogous to that of Lemma 9 and is omitted. □

Lemma 25. *For all $i, j, \mathbb{C}, \mathbb{C}', \mathbb{C}'', \sigma, \sigma', \sigma'', \epsilon$, if $\mathbb{C}, \sigma, \rightsquigarrow^i \mathbb{C}'', \sigma'', ok$ and $\mathbb{C}'', \sigma'' \rightsquigarrow^j \mathbb{C}', \sigma', \epsilon$, then $\mathbb{C}, \sigma \rightsquigarrow^{i+j} \mathbb{C}', \sigma', \epsilon$.*

Proof. The proof of this lemma is analogous to that of Lemma 10 and is omitted. □

Lemma 26. *For all p, C, if $\vdash [p]\ \mathsf{C}\ [\infty]$ can be derived using the proof rules in Fig. 10, then:*

$$\begin{aligned}&\forall \sigma_p \in p.\ \forall \sigma.\ \sigma_p \ \# \ \sigma \implies \\ &\qquad \exists \mathsf{C}_1, \sigma_1, \mathsf{C}_2, \sigma_2, \cdots .\ \mathsf{C}, \sigma_p \circ \sigma \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots\end{aligned}$$

Proof. By induction on the structure of the divergence rules in Fig. 2 and Fig. 10.

Case Div-Seq1

Pick arbitrary p, C_1, C_2 such that $[p]\ \mathsf{C}_1; \mathsf{C}_2\ [\infty]$. Pick an arbitrary $\sigma_p \in p$ and σ such that $\sigma_p \ \# \ \sigma$. From the $[p]\ \mathsf{C}_1\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series

$C'_2, C'_3, \cdots$, and $\sigma_2, \sigma_3, \cdots$, such that $C_1, \sigma_p \circ \sigma \rightsquigarrow^+ C'_2, \sigma_2, ok \rightsquigarrow^+ C'_3, \sigma_3, ok \rightsquigarrow^+ \cdots$. As such, from the definition of $\rightsquigarrow^+$ and Lemma 23 we have $C_1; C_2, \sigma_p \circ \sigma \rightsquigarrow^+ C'_2; C_2, \sigma_2, ok \rightsquigarrow^+ C'_3; C_2, \sigma_3, ok \rightsquigarrow^+ \cdots$, as required.

Case 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 σ such that $\sigma_p \ \#\ \sigma$. From the $\vdash_B [p]\ C_1\ [ok\colon q]$ premise and Lemma 18 we know there exists $\sigma_q \in q$ and n such that $C_1, \sigma_p \circ \sigma \xrightarrow{n} -, \sigma_q \circ \sigma, ok$. Moreover, since $\sigma_q \in q$, from the $[q]\ C_2\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $C'_3, C'_4, \cdots$ and $\sigma_3, \sigma_4, \cdots$, such that $C_2, \sigma_q \circ \sigma \rightsquigarrow^+ C'_3, \sigma_3, ok \rightsquigarrow^+ C'_4, \sigma_4, ok \rightsquigarrow^+ \cdots$. As $C_1, \sigma_p \circ \sigma \xrightarrow{n} -, \sigma_q \circ \sigma, ok$ and $C_2, \sigma_q \circ \sigma \rightsquigarrow^+ C'_3, \sigma_3, ok$, from the definition of $\rightsquigarrow^+$ and Lemma 24 we have $C_1; C_2, \sigma_p \circ \sigma \rightsquigarrow^+ C'_3, \sigma_3, ok$. Moreover, as $C'_3, \sigma_3 \rightsquigarrow^+ C'_4, s_4, ok \rightsquigarrow^+ \cdots$, we have $C_1; C_2, \sigma_p \circ \sigma \rightsquigarrow^+ C'_3, \sigma_3, ok \rightsquigarrow^+ C'_4, \sigma_4, ok \rightsquigarrow^+ \cdots$, as required.

Case 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 σ such that $\sigma_p \ \#\ \sigma$. From the $[p]\ C_i\ [\infty]$ premise and the inductive hypothesis we know there exists an infinite series $C_3, C_4, \cdots$ and $\sigma_3, \sigma_4, \cdots$, such that $C_i, \sigma_p \circ \sigma \rightsquigarrow^+ C_3, \sigma_3, ok \rightsquigarrow^+ C_4, \sigma_4, ok \rightsquigarrow^+ \cdots$. Moreover, from SL-Choice we have $C_1 + C_2, \sigma_p \circ \sigma \longrightarrow C_i, \sigma_p \circ \sigma, ok$. And thus we have $C_1 + C_2, \sigma_p \circ \sigma \longrightarrow C_i, \sigma_p \circ \sigma, ok \rightsquigarrow^+ C_3, \sigma_3, ok \rightsquigarrow^+ C_4, \sigma_4, ok \rightsquigarrow^+ \cdots$. That is, by definition of $\rightsquigarrow^+$ we have $C_1 + C_2, \sigma_p \circ \sigma \rightsquigarrow^+ C_3, \sigma_3, ok \rightsquigarrow^+ C_4, \sigma_4, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-LoopUnfold
Pick arbitrary p, C such that $[p]\ C^\star\ [\infty]$. Pick an arbitrary $\sigma_p \in p$ and σ such that $\sigma_p \ \#\ \sigma$. From the $[p]\ C; C^\star\ [\infty]$ 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 $C; C^\star, \sigma_p \circ \sigma \rightsquigarrow^+ C_1, \sigma_1, ok \rightsquigarrow^+ C_2, \sigma_2, ok \rightsquigarrow^+ \cdots$. Moreover, from SL-Loop we have $C^\star, \sigma_p \circ \sigma \longrightarrow C; C^\star, \sigma_p \circ \sigma, ok$. And thus we have $C^\star, \sigma_p \circ \sigma \longrightarrow C; C^\star, \sigma_p \circ \sigma, ok \rightsquigarrow^+ C_1, \sigma_1, ok \rightsquigarrow^+ C_2, \sigma_2, ok \rightsquigarrow^+ \cdots$. That is, by definition of $\rightsquigarrow^+$ we have $C^\star, \sigma_p \circ \sigma \rightsquigarrow^+ C_1, \sigma_1, ok \rightsquigarrow^+ C_2, \sigma_2, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-LoopNest
This rule can be derived as follows:

$$\dfrac{\dfrac{[p]\ C\ [\infty]}{[p]\ C; C^\star\ [\infty]}\ \textsc{Div-Seq1}}{[p]\ C^\star\ [\infty]}\ \textsc{Div-LoopUnfold}$$

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 $\vdash [p]\ C^\star\ [\infty]$. From SL-Loop we then have:

$$\forall \sigma_p \in p, \sigma.\ \sigma_p \ \#\ \sigma \Rightarrow C^\star, \sigma_p \circ \sigma \longrightarrow C; C^\star, \sigma_p \circ \sigma, ok \tag{15}$$

From the $\vdash_B [p]\ C\ [ok\colon q]$ premise, Lemma 18, and the $q \subseteq p$ premise we know $\forall \sigma_p \in p, \sigma.\ \sigma_p \ \#\ \sigma \Rightarrow \exists \sigma'_p \in p, n.\ C, \sigma_p \circ \sigma \xrightarrow{n} -, \sigma'_p \circ \sigma, ok$ and thus from Lemma 14 $C, \sigma_p \circ \sigma \xrightarrow{n} \mathsf{skip}, \sigma'_p \circ \sigma, ok$.

That is, from the axiom of choice we know there exist $f : p \to p$ and $g : p \to \mathbb{N}$ such that:

$$\forall \sigma_p \in p, \sigma.\ \sigma_p \# \sigma \Rightarrow \mathrm{C}, \sigma_p \circ \sigma \xrightarrow{g(\sigma_p)} \mathrm{skip}, f(\sigma_p) \circ \sigma, ok \land f(\sigma_p) \in p \tag{16}$$

In what follows, we show that $\forall \sigma_p \in p, \sigma.\ \sigma_p \# \sigma \Rightarrow \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$.

Pick an arbitrary $\sigma_p \in p$ and σ such that $\sigma_p \# \sigma$. From (3) we have $\mathrm{C}, \sigma_p \circ \sigma \xrightarrow{g(\sigma_p)} \mathrm{skip}, f(\sigma_p) \circ \sigma, ok$. There are now two cases to consider: i) $g(\sigma_p) = 0$; or ii) $g(\sigma_p) > 0$. In case (i), we then have $\mathrm{C} = \mathrm{skip}$ and $\sigma_p = f(\sigma_p)$. As such, from SL-SeqSkip we have $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \to \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$, and thus by definition of $\rightsquigarrow^1$ we have $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^1 \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$

In case (ii), from $\mathrm{C}, \sigma_p \circ \sigma \xrightarrow{g(\sigma_p)} \mathrm{skip}, f(\sigma_p) \circ \sigma, ok$ and Lemma 22 we have $\mathrm{C}, \sigma_p \circ \sigma \rightsquigarrow^{g(\sigma_p)} \mathrm{skip}, f(\sigma_p) \circ \sigma, ok$. Consequently, from Lemma 23 we have $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^{g(s)} \mathrm{skip}; \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$. On the other hand, from SL-SeqSkip we have $\mathrm{skip}; \mathrm{C}^\star, f(\sigma_p) \circ \sigma \to \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$ and thus by definition of $\rightsquigarrow^1$ we have $\mathrm{skip}; \mathrm{C}^\star, f(\sigma_p) \circ \sigma \rightsquigarrow^1 \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$. From Lemma 25, $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^{g(s)} \mathrm{skip}; \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$ and $\mathrm{skip}; \mathrm{C}^\star, f(\sigma_p) \circ \sigma \rightsquigarrow^1 \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$ we know there exists i such that $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^i \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$.

That is, in both cases we know there exists i such that $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^i \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$. As such, from (15) and the definition of $\rightsquigarrow^{i+1}$ we have $\mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^{i+1} \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$, i.e. $\mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok$. That is, from (16) we have:

$$\forall \sigma_p \in p, \sigma.\ \sigma_p \# \sigma \Rightarrow \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok \land f(\sigma_p) \in p \tag{17}$$

Pick an arbitrary $\sigma_p \in p$ and σ such that $\sigma_p \# \sigma$. From (17) we then know $\mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathrm{C}^\star, f(\sigma_p) \circ \sigma, ok \rightsquigarrow^+ \mathrm{C}^\star, f^2(\sigma_p) \circ \sigma, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-Subvar

Pick arbitrary p, C, q such that $\vdash [p(0)]\ \mathrm{C}^\star\ [\infty]$. From SL-Loop we then have:

$$\forall \sigma_p \in p, \sigma.\ \sigma_p \# \sigma \Rightarrow \mathrm{C}^\star, \sigma_p \circ \sigma \to \mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma, ok \tag{18}$$

From the $\forall n \in \mathbb{N}.\ \vdash_{\mathrm{B}} [p(n)]\ \mathrm{C}\ [ok\colon p(n{+}1)]$ premise, Lemma 18, and the $q \subseteq p$ premise we know $\forall n \in \mathbb{N}, \sigma_p \in p(n), \sigma.\ \sigma_p \# \sigma \Rightarrow \exists \sigma'_p \in p(n{+}1), k.\ \mathrm{C}, \sigma_p \circ \sigma \xrightarrow{k} -, \sigma'_p \circ \sigma, ok$. That is, from the axiom of choice we know there exists a series of functions, $f_1, g_1, f_2, g_2 \cdots$ such that for each $i \in \mathbb{N}$, we have $f_i : p(i{-}1) \to p(i)$ and $g_i : p(i{-}1) \to \mathbb{N}$ such that:

$$\forall i \in \mathbb{N}^+.\ \forall \sigma_p \in p(i-1), \sigma.\ \sigma_p \# \sigma \Rightarrow \mathrm{C}, \sigma_p \circ \sigma \xrightarrow{g_i(\sigma_p)} \mathrm{skip}, f_i(\sigma_p) \circ \sigma, ok \land f_i(\sigma_p) \in p(i) \tag{19}$$

In what follows, we show that $\forall i \in \mathbb{N}^+.\ \forall \sigma_p \in p(i{-}1), \sigma.\ \sigma_p \# \sigma \Rightarrow \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$.

Pick an arbitrary $i \in \mathbb{N}^+$, $\sigma_p \in p(i{-}1)$ and σ such that $\sigma_p \# \sigma$. From (19) we have $\mathrm{C}, \sigma_p \circ \sigma \xrightarrow{g_i(\sigma_p)} \mathrm{skip}, f_i(\sigma_p) \circ \sigma, ok$. There are now two cases to consider: a) $g_i(\sigma_p) = 0$; or b) $g_i(\sigma_p) > 0$. In case (a), we then have $\mathrm{C} = \mathrm{skip}$ and $\sigma_p = f_i(\sigma_p)$. As such, from SL-SeqSkip we have $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \to \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$, and thus by definition of $\rightsquigarrow^1$ we have $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^1 \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$.

In case (b), from $\mathrm{C}, \sigma_p \circ \sigma \xrightarrow{g_i(\sigma_p)} \mathrm{skip}, f_i(\sigma_p) \circ \sigma, ok$ and Lemma 22 we have $\mathrm{C}, \sigma_p \circ \sigma \rightsquigarrow^{g_i(\sigma_p)} \mathrm{skip}, f_i(\sigma_p) \circ \sigma, ok$. Consequently, from Lemma 23 we have $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^{g_i(\sigma_p)} \mathrm{skip}; \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$. On the other hand, from SL-SeqSkip we have $\mathrm{skip}; \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma \to \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$ and thus by definition of $\rightsquigarrow^1$ we have $\mathrm{skip}; \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma \rightsquigarrow^1 \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$. From Lemma 25, $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^{g_i(\sigma_p)} \mathrm{skip}; \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$ and $\mathrm{skip}; \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma \rightsquigarrow^1 \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$ we know there exists j such that $\mathrm{C}; \mathrm{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^j \mathrm{C}^\star, f_i(\sigma_p) \circ \sigma, ok$.

That is, in both cases we know there exists j such that $\mathsf{C}; \mathsf{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^j \mathsf{C}^\star, f_i(\sigma_p) \circ \sigma, ok$. As such, from (18) and the definition of $\rightsquigarrow^{j+1}$ we have $\mathsf{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^{j+1} \mathsf{C}^\star, f_i(\sigma_p) \circ \sigma, ok$, i.e. $\mathsf{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathsf{C}^\star, f_i(\sigma_p) \circ \sigma, ok$. That is, from (19) we have:

$$\forall i \in \mathbb{N}^+.\ \forall \sigma_p \in p(i{-}1), \sigma.\ \sigma_p \# \sigma \Rightarrow \mathsf{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathsf{C}^\star, f_i(\sigma_p) \circ \sigma, ok \land f_i(\sigma_p) \in p(i) \tag{20}$$

Pick an arbitrary $\sigma_p \in p(0)$ and σ such that $\sigma_p \# \sigma$. From (20) we then know $\mathsf{C}^\star, \sigma_p \circ \sigma \rightsquigarrow^+ \mathsf{C}^\star, f_1(\sigma_p) \circ \sigma, ok \rightsquigarrow^+ \mathsf{C}^\star, f_2(\sigma_p) \circ \sigma, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-Cons
Pick arbitrary p, C such that $\vdash [p]\ \mathsf{C}\ [\infty]$. Pick an arbitrary $\sigma_p \in p$ and σ such that $\sigma_p \# \sigma$. From the $p \subseteq p'$ premise we know $\sigma_p \in p'$. As such, from the $[p']\ \mathsf{C}\ [\infty]$ premise we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $\sigma_1, \sigma_2, \cdots$, such that $\mathsf{C}, \sigma_p \circ \sigma \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-Disj
Pick arbitrary p, C such that $\vdash [p]\ \mathsf{C}\ [\infty]$. Pick an arbitrary $\sigma_p \in p_1 \lor p_2$ and σ such that $\sigma_p \# \sigma$. That is, $s \in p_i$ where either $i = 1$ or $i = 2$. As such, from the $\vdash [p_i]\ \mathsf{C}\ [\infty]$ premises we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $\sigma_1, \sigma_2, \cdots$, such that $\mathsf{C}, \sigma_p \circ \sigma \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$, as required.

Case Div-Frame
Pick arbitrary p, r, C such that $[p * r]\ \mathsf{C}\ [\infty]$. Pick an arbitrary $\sigma_{pr} \in p * r$ and σ such that $\sigma_{pr} \# \sigma$. As $\sigma_{pr} \in p * r$, 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 σ_{pr} and since $\sigma_{pr} \# \sigma$ we know $\sigma_r \# \sigma$ and $\sigma_p \# \sigma_r \circ \sigma$.

On the other hand, from the premise of Div-Frame we have $[p]\ \mathsf{C}\ [\infty]$ and thus from the inductive hypothesis we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $\sigma_1, \sigma_2, \cdots$, such that $\mathsf{C}, \sigma_p \circ (\sigma_r \circ \sigma) \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$. That is, by associativity of $\circ$ we have $\mathsf{C}, (\sigma_p \circ \sigma_r) \circ \sigma \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$, i.e. $\mathsf{C}, \sigma_{pr} \circ \sigma \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$, as required. □

Lemma 27. *For all p, C, if $\vdash [p]\ \mathsf{C}\ [\infty]$ is derivable using the rules in Fig. 2 and Fig. 10, then $\models [p]\ \mathsf{C}\ [\infty]$ holds.*

Proof. Pick arbitrary p, C such that $[p]\ \mathsf{C}\ [\infty]$ is derivable using the rules in Fig. 2 and Fig. 10. Pick an arbitrary $\sigma_p = (s_p, h_p) \in p$. It then suffices to show there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $\sigma_1, \sigma_2, \cdots$, such that $\mathsf{C}, \sigma_p \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$.

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 26 we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $\sigma_1, \sigma_2, \cdots$, such that $\mathsf{C}, \sigma_p \circ \sigma_0 \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$. That is, as $\sigma_p \circ \sigma_0 = \sigma_p$, we know there exists an infinite series $\mathsf{C}_1, \mathsf{C}_2, \cdots$ and $\sigma_1, \sigma_2, \cdots$, such that $\mathsf{C}, \sigma_p \rightsquigarrow^+ \mathsf{C}_1, \sigma_1, ok \rightsquigarrow^+ \mathsf{C}_2, \sigma_2, ok \rightsquigarrow^+ \cdots$, as required. □

Theorem 19 (UNTer$^{\text{SL}}$ soundness). *For all p, q, C and ϵ:*

1) *if $\vdash_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10, then $\models_{\mathsf{B}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds;*
2) *if $\vdash_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 10, then $\models_{\mathsf{F}} [p]\ \mathsf{C}\ [\epsilon : q]$ holds; and*
3) *if $\vdash [p]\ \mathsf{C}\ [\infty]$ is derivable using the rules in Fig. 10, then $\models [p]\ \mathsf{C}\ [\infty]$ holds.*

Proof. The proof of part (1) follows immediately from Lemma 19. The proof of part (2) follows immediately from Lemma 21. The proof of part (3) follows immediately from Lemma 27. □

H Non-Termination CVEs

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

Table 4. 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;
                break;
            }
        }
    }
```

H.2 Web software: log4j (Java, CVE 2021-45105)

Table 5. 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());
(...)
}
```

H.3 Data mining Software: GraphQL (Golang, Sept 2022)

Table 6. 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)
   {
       // FIXME: comment below 4 lines
      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()]
      // FIX: replace above 2 lines with:
      //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)
}
```

H.4 System Software: Linux Kernel (C, CVE-2020-25641)

Table 7. Termination bug in the Linux kernel (August 2020). Macro for_each_bvec contains an infinite loop due to zero sized bvec which fails to increment the loop index. Bug discussed at https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg2262077.html. Details available at https://nvd.nist.gov/vuln/detail/CVE-2020-25641. Table shows minimized vulnerable code.

```
+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)))
```

H.5 Graphical Software : Blender (C language)

Table 8. 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.

```
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;
        }
    }
}
```

H.6 Machine Learning Software : Sklearn (Python)

Table 9. 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 outter one. Fix available at https://github.com/scikit-learn/scikit-learn/pull/21271/commits/325d32fedb48b42faa32b0873a9eeee9ff35a125. Table shows minimized vulnerable code.

```
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
```

H.7 Cryptographic Software: OpenSSL (C lang, CVE-2022-0778)

Table 10. 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).

```
-       /* 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;
        }
```

Received 2024-04-06; accepted 2024-08-18