

Exact Separation Logic

Towards bridging the gap between verification and bug-finding

PETAR MAKSIMOVIĆ, Imperial College London, United Kingdom

CAROLINE CRONJÄGER, Ruhr-Universität Bochum, Germany

JULIAN SUTHERLAND, Imperial College London, United Kingdom

ANDREAS LÖÖW, Imperial College London, United Kingdom

PHILIPPA GARDNER, Imperial College London, United Kingdom

Over-approximating (OX) program logics, such as separation logic, are used to verify properties of heap-manipulating programs: all terminating behaviour is characterised but the reported results and errors need not be reachable. OX function specifications are thus incompatible with true bug-finding supported by symbolic execution tools such as Pulse and Gillian. In contrast, under-approximating (UX) program logics, such as incorrectness separation logic, are used to find true results and bugs: reported results and errors are reachable, but not all behaviour can be characterised. UX function specifications thus cannot capture full verification.

We introduce exact separation logic (ESL), which provides fully verified function specifications compatible with true bug finding: all terminating behaviour is characterised, and all reported results and errors are reachable. ESL requires subtle definitions of internal and external function specifications compared with the familiar definitions of OX logics. It supports reasoning about mutually recursive functions and non-termination. We prove frame-preserving soundness for ESL, demonstrating, for the first time, functional compositionality for a non-OX program logic. We investigate the expressivity of ESL and the role of abstraction in UX reasoning by verifying abstract ESL specifications of list algorithms. To show overall viability of exact verification for true bug-finding, we formalise a compositional symbolic execution semantics capable of using ESL specifications and characterise the conditions that these specifications must respect so that true bug-finding is preserved.

1 INTRODUCTION

Program logics were introduced for reasoning about program correctness, originating with Hoare logic [28] and evolving to separation logics (SL) [6, 38, 43] for reasoning in a functionally compositional way about heap-manipulating programs. These over-approximating (OX) logics are well-suited for verifying properties of programs: OX specifications capture all terminating behaviour, non-termination can also be captured, in some scenarios, via the postcondition False, but not all reported results and errors are necessarily reachable.

By contrast, under-approximating (UX) logics were comparatively recently introduced for finding true results and bugs, originating with reverse Hoare logic (RHL) [13] for reasoning about probabilistic programs and coming to prominence with incorrectness logic (IL) [37] for reasoning about program incorrectness: UX specifications capture some terminating behaviour, non-termination cannot be characterised, and all reported results and errors are reachable. Since then, many UX logics have been introduced, including incorrectness separation logic (ISL) [40], concurrent incorrectness separation logic (CISL) [41], and insecurity separation logic (InsecSL) [35].

The application of UX reasoning to program incorrectness arose from the challenge to record, as function summaries, the true results and bugs coming from Meta’s symbolic execution tool, Pulse [37, 40]. Such information tends to be partial: e.g., the SAT solver can fail; the loop unrolling fuel can run out; a function may not be found; or a computation may take too long. This partiality is embedded into the meaning of UX specifications, but, as a consequence, UX specifications do not provide verification guarantees. In addition, symbolic execution tools are not able to use OX specifications without breaking the UX guarantee of no false positives. Our goal is to develop fully verified function specifications of, for example, data-structure libraries that are compatible with symbolic execution tools that target true bug-finding.

We introduce exact separation logic (ESL) for reasoning about heap-manipulating sequential programs, using a simple demonstrator programming language with a linear memory model. We demonstrate that ESL provides verified exact (EX) specifications compatible with true bug-finding. These specifications capture all terminating behaviour in full, all reported results and errors are reachable, and non-termination can be characterised using the post-condition `False`. Despite these strong properties, we have found ESL specifications to be more expressive and ESL proofs to be easier than one might expect. We illustrate this by verifying EX correctness specifications of standard list algorithms and finding standard language errors associated with examples from the UX literature. In addition, we adapt Gillian [16, 20, 33, 34] to EX verification, and verify these specifications semi-automatically. We prove a frame-preserving soundness result for ESL with mutually recursive functions, presented in such a way that it is immediately transferable to ISL and SL, demonstrating, for the first time, functional compositionality of UX reasoning. In addition, we introduce a symbolic execution semantics that can call functions with ESL specifications, and prove a correctness result that demonstrates that verified ESL function specifications are indeed compatible with true bug-finding. Finally, through the list-algorithm examples, we investigate the interaction of abstraction with UX reasoning, highlighting the difference between abstraction and over-approximation, and delineate a boundary for the usability of abstraction in true bug-finding.

In order to better understand the difference between ESL and ISL, consider the ISL quadruple $[P] \ C \ [ok : Q_{ok}] \ [err : Q_{err}]$, which tells us that any state satisfying either the success post-condition Q_{ok} or the error post-condition Q_{err} is reachable from some state satisfying the pre-condition P by executing the command C . It says nothing about any other behaviours of C , and even if the post-condition is complete, it is not possible to identify this within ISL. In contrast, an ESL quadruple $(P) \ C \ (ok : Q_{ok}) \ (err : Q_{err})$ additionally states that all terminating executions of C starting from a state satisfying the pre-condition either end successfully in a state that satisfies Q_{ok} or fault in a state that satisfies Q_{err} . These exact quadruples, which provide unified reasoning about correctness and incorrectness, are fundamental to ESL and to OX logics requiring such a treatment of errors.¹ They are not fundamental to UX logics, in that a UX quadruple can be split into two separate triples.

An important property of OX separation logics is that the reasoning is functionally compositional, and hence scalable.² This function compositionality is the reason why Meta’s Infer and Pulse tools can work on industrial-scale codebases. This property is not immediate for ESL and UX logics, as the OX definitions do not transfer and, to our knowledge, functional compositionality for UX logics has not been studied.³ We develop ESL with mutually recursive functions, requiring subtle definitions of *external* function specifications, which provide the interface that the function exposes to the client, and *internal* function specifications, which provide the interface to the function’s implementation. With OX logics, these specifications are well-understood and the gap between them is small. For ESL and UX logics, this gap is larger, especially with the post-conditions. This is because information cannot be lost within the reasoning, and so parameters and local variables must remain in the internal post-condition, but at the same time must not be present in the external post-condition, as their scope does not extend outside the function.

We present ESL rules for calling functions and for extending the set of available functions (the environment) with a new cluster of mutually recursive functions: function call is analogous to its OX counterpart; environment extension is substantially different. First, it captures the more intricate transfer from the internal to the external post-condition. Second, when a new cluster is added, terminating and non-terminating specifications are treated separately. In particular, to

¹Strictly speaking, the ESL post-condition could be expressed as a disjunction of *ok*- and *err*-labelled assertions, but the quadruple distinction is helpful as compound commands (e.g., sequence) treat the two differently.

²Interestingly, we were not able to find a direct proof of functional compositionality for sequential SL.

³ISL [40] and InsecSL [35] use function specifications in examples and associated tools, but provide no logic rules.

preserve UX reachability, all terminating specifications must be provably terminating: this we achieve by imposing a joint measure on the pre-conditions and restricting (mutually) recursive calls to pre-conditions of smaller measure only. Non-terminating specifications, on the other hand, may be used without constraints to prove themselves, as in OX logics. Relying on transfinite and Scott induction [45], we prove a frame-preserving soundness result for ESL, achieving functional compositionality. Our approach can be simply adapted to ISL and SL. We believe we are the first to have demonstrated functional compositionality for UX reasoning.

We illustrate the usability of ESL by exploring singly-linked list algorithms and specifications with different degrees of abstraction. For example, two specifications of the list-length algorithm are:

$$\begin{aligned} (x = x \star \text{list}(x, n)) \text{ LLen}(x) \quad (\text{list}(x, n) \star \text{ret} = n) & \quad (\text{SP1}) \\ (x = x \star \text{list}(x, xs, vs)) \text{ LLen}(x) \quad (\text{list}(x, xs, vs) \star \text{ret} = |vs|) & \quad (\text{SP2}) \end{aligned}$$

where the triple notation means that no errors occur.⁴ In (SP1), $\text{list}(x, n)$ denotes the standard list abstraction that tells us that in memory, starting from address x , there is a singly-linked list of length n , hiding information about node addresses and values. This shows that ESL specifications can be as abstract as their OX counterparts, which may seem counter-intuitive given the UX requirement of not losing any information. The insight is that information hidden via abstraction in the pre-condition may soundly remain hidden in the post-condition as well. We observe that specifications such as (SP1) cannot be used in standard symbolic execution precisely due to this hiding of information. By contrast, the specification (SP2) features the $\text{list}(x, xs, vs)$ abstraction, which provides full information about the list structure through variable xs , denoting the list of node addresses, and vs , denoting a list of values.⁵ With respect to (SP1), it additionally captures the fact that the list structure does not change, and can therefore be used in standard symbolic execution for true bug-finding. Abstractions that expose node addresses are also needed for specifying, e.g., a list-free algorithm, where the post-condition must explicitly state that these addresses have been freed, as resource cannot be forgotten with EX or UX reasoning. We adapt the semi-automatic OX verification of Gillian [33] to handle exact verification for recursive functions, and prove a number of exact list-algorithm specifications with varying degrees of abstraction.

We introduce compositional symbolic execution (CSE) for our demonstrator language, including a function call rule that allows it to use EX and UX specifications. We prove that if the abstractions used in the specifications are strictly exact, then our CSE satisfies backward completeness, a property of symbolic execution analogous to UX validity in program logics. This result is the first to demonstrate the feasibility of combining true bug-finding with verified separation-logic specifications.

Contributions. In summary, the contributions of this paper are:

- the introduction of exact specifications and ESL, a sound exact separation logic for verifying such specifications (§4);
- the first proof of sound function compositionality for a non-OX program logic (§4.12);
- exact specifications of a number of illustrative examples, demonstrating how ESL can be used to reason about data-structure libraries, language errors, mutual recursion, and non-termination (§5);
- the adaptation of the Gillian platform to support EX verification of recursive functions (§5);
- a compositional symbolic execution semantics for true bug-finding, which can soundly use verified ESL/ISL function specifications (§6);
- an investigation into the use of abstraction for true bug-finding in UX program logics and symbolic execution (§5, §6).

⁴In ESL, an equality assertion, $E_1 = E_2$, denotes that the heap is empty and that E_1 and E_2 represent the same value.

⁵Abstractions such as $\text{list}(x, xs, vs)$ are called strictly exact in the literature [46, p. 149]. Their property is that, for any concretisation of their parameters, there exists at most one heap that satisfies such a concretisation.

2 RELATED WORK

We place ESL in the context of related work on OX and UX logics, symbolic analysis, and associated tools. In particular, we highlight its relationship to Gillian [16, 20, 33, 34], a recently-developed platform for unified symbolic analysis. In addition, we briefly discuss existing approaches to use of summaries and abstraction in symbolic execution.

OX Program Logics. OX separation logics have been applied to many diverse languages and libraries, including Java and `java.util.concurrent` [11, 15, 39], C [3, 29, 31], POSIX [36], JavaScript and DOM [21, 22, 42], Wasm [44], and Rust [4, 30]. These works account at times for language errors, almost always use exact axioms for the basic commands, but are set in the overall context of OX reasoning. An interesting question is just how much of their OX reasoning can be made exact; our examples, which use list abstractions, suggest that this percentage could be high.

The combination of ideas originating from SL, together with the bi-abductive technique needed for automation [9, 10], led first to the Monoidics Infer tool for verification and then to Meta’s Infer tool for bug finding [8]. However, the specifications generated by both of these tools are OX in principle, meaning that the reported bugs are not necessarily true.

UX Program Logics. The work on UX program logics originated with RHL [13], which introduced backward consequence and UX correctness triples, and was used to prove properties of non-deterministic algorithms, e.g., an array shuffle. The recent work on IL [37], ISL [40], CISL [41], and InsecSL [35] shifted the focus of UX reasoning to true bug-finding, expressed through various UX incorrectness triples. The motivation for IL and ISL came from Meta’s bug-finding tool, Pulse [37, 40], developed by the Infer team with the goal of only generating true bugs. InsecSL also comes with an accompanying symbolic execution tool, and the CISL framework has formalised ideas arising from the concurrent analysis in the work on RacerD [26] and Views [14].

To our knowledge, none of this work on UX reasoning addresses functions, despite functional compositionality being essential for scalability. Our approach to internal and external function specifications, informed by the backward consequence of UX logics and hinted at in InsecSL, yields a unified treatment of functions across OX and UX reasoning. Our ESL soundness proof, which implies functional compositionality, transfers to ISL by removing the non-terminating specifications and the Scott-induction part of the proof. When it comes to abstraction, only RHL uses recursive predicates (e.g., for list permutation), but their predicates are first-order and not abstract.

Independently, LCL_A [7] is a non-functionally-compositional, first-order logic that combines UX and OX reasoning using abstract interpretation. It is parametric on an abstract domain A , and proves UX triples of the form $\vdash_A [P] \text{ C } [Q]$ where, under certain conditions, the triple also guarantees verification. These conditions, however, normally mean that only a limited number of pre-conditions can be handled. The conditions also have to be checked per-command and if they fail to hold (due to, e.g., issues with Boolean guards, which are known to be a major source of incompleteness), then the abstract domain has to be incrementally adjusted; the complexity of this adjustment and the expressivity of the resulting formalism is unclear.

Gillian and ESL. Our motivation for ESL came from Gillian [16, 20, 33, 34], a recently-introduced multi-language platform for unified symbolic analysis that uses EX and OX function specifications. Gillian’s core symbolic execution engine [34], used for whole- program symbolic testing, has been shown to satisfy forward soundness and backward completeness, properties that strongly correspond to the OX and UX parts of our ESL soundness result. Gillian’s SL-based compositional verification provides OX specifications [20, 33], and its compositional symbolic testing based on bi-abduction [20] generates (mostly) EX specifications.

Compositional Symbolic Execution. There exists a substantial body of work on symbolic execution with function summaries (e.g. [1, 23, 25, 27, 32, 47]), most of which is based on first-order logic. We highlight the work of Godefroid et al., which initially used first-order exact summaries of bounded program behaviour to drive the compositional dynamic test generation of SMART [23], and later distinguished between may (OX) and must (UX) summaries, leveraging the interaction between them to design the SMASH algorithm for compositional property checking and test generation [25]. Our ESL specifications are able to capture properties of unbounded program behaviour, as well as non-termination, and can be used in both OX and UX program-logic reasoning. Our compositional symbolic execution is able to soundly use ISL/ESL specifications, which can contain arbitrary information about the heap. When it comes to abstraction, for example, Anand et al. [2] implement linked-list and array abstractions for true bug-finding in non-compositional symbolic execution, in the context of the Java PathFinder, and use it to find bugs in list and array partitioning algorithms. True bug-finding is maintained by checking for state subsumption, which requires code modification rather than annotation and a record of all previously visited states.

3 THE PROGRAMMING LANGUAGE

We introduce ESL using a simple programming language, highlighting the most important aspects in the body of the paper, and delegating the rest to Appendix A due to space constraints.

Language Syntax

$$\begin{aligned}
 v \in \text{Val} &::= n \in \text{Nat} \mid b \in \text{Bool} \mid s \in \text{Str} \mid \text{null} \mid \bar{v} & x \in \text{PVar} \\
 E \in \text{PExp} &::= v \mid x \mid E + E \mid E - E \mid \dots \mid E = E \mid E < E \mid \neg E \mid E \wedge E \mid \dots \mid E : E \mid E \cdot E \mid \dots \\
 C \in \text{Cmd} &::= \text{skip} \mid x := E \mid x := \text{nondet} \mid \text{error}(E) \mid \text{if } (E) C \text{ else } C \mid \text{while } (E) C \mid C; C \mid \\
 & \quad y := f(\vec{E}) \mid x := [E] \mid [E] := E \mid x := \text{new}(E) \mid \text{free}(E)
 \end{aligned}$$

Syntax. The language syntax is given above. Values, $v \in \text{Val}$, include: natural numbers, $n \in \text{Nat}$; Booleans, $b \in \text{Bool} \triangleq \{\text{true}, \text{false}\}$; strings, $s \in \text{Str}$; a dedicated value null ; and lists of values, $\bar{v} \in \text{List}$. Expressions, $E \in \text{Exp}$, comprise values, program variables, $x \in \text{PVar}$, and various unary and binary operators (e.g., addition, equality, negation, conjunction, list prepending, and list concatenation). Commands comprise: variable assignment; non-deterministic number generation; error raising; if statement; while loop; command sequencing; function call; and memory management commands, that is, lookup, mutation, allocation, and deallocation. The sets of program variables for expressions and commands, denoted by $\text{pv}(E)$ and $\text{pv}(C)$ respectively, and the sets of modified variables for commands, denoted by $\text{mod}(C)$, are defined in the standard way.

Definition 3.1 (Functions). A function, denoted by $f(\vec{x}) \{C; \text{return } E\}$, comprises: a function identifier, $f \in \text{Fid}$, given by a string; the function parameters, \vec{x} , given by a list of distinct program variables; a function body, $C \in \text{Cmd}$; and a return expression, $E \in \text{PExp}$, with $\text{pv}(E) \subseteq \{\vec{x}\} \cup \text{pv}(C)$.

Program variables in function bodies that are not the function parameters are treated as local variables initialised to null , with their scope not extending beyond the function.

Definition 3.2 (Function Implementation Contexts). A function implementation context γ is a finite partial function from function identifiers to their implementations:

$$\gamma : \text{Fid} \rightarrow_{\text{fin}} \text{PVar List} \times \text{Cmd} \times \text{PExp}$$

For $\gamma(f) = (\vec{x}, C, E)$, we also write $f(\vec{x})\{C; \text{return } E\} \in \gamma$.

Operational Semantics. We define an operational semantics that gives a complete account of the behaviour of commands and does not get stuck on any input, as we explicitly account for language errors and missing resource errors.

Definition 3.3 (Stores, Heaps, States). Variable stores (also: stores), $s : \text{PVar} \rightarrow_{\text{fin}} \text{Val}$, are partial finite functions from program variables to values. Heaps, $h : \text{Nat} \rightarrow_{\text{fin}} (\text{Val} \uplus \emptyset)$, are partial finite functions from natural numbers to values extended with a dedicated symbol $\emptyset \notin \text{Val}$. Program states (also: states), $\sigma = (s, h)$, consist of a store and a heap.

Heaps are used to model the memory, and the dedicated symbol $\emptyset \notin \text{Val}$ is required for UX frame preservation⁶ to hold (cf. Definition 4.5). In particular, $h(n) = v$ means that an allocated heap cell with address n contains the value v ; and $h(n) = \emptyset$ means that a heap cell with address n has been deallocated [17–19, 21, 40]. This linear memory model is used in much of the SL literature, including ISL [40]. Onward, \emptyset denotes the empty heap, $h_1 \uplus h_2$ denotes heap disjoint union, and $h_1 \# h_2$ denotes that h_1 and h_2 are disjoint.

Definition 3.4 (Expression Evaluation). The evaluation of an expression E with respect to a store s , denoted $\llbracket E \rrbracket_s$, results in either a value or a dedicated symbol denoting an evaluation error, $\downarrow \notin \text{Val}$. Some illustrative cases are:

$$\llbracket v \rrbracket_s = v \quad \llbracket x \rrbracket_s = \begin{cases} s(x), & x \in \text{dom}(s) \\ \downarrow, & \text{otherwise} \end{cases} \quad \llbracket E_1 + E_2 \rrbracket_s = \begin{cases} \llbracket E_1 \rrbracket_s + \llbracket E_2 \rrbracket_s, & \llbracket E_1 \rrbracket_s, \llbracket E_2 \rrbracket_s \in \text{Nat} \\ \downarrow, & \text{otherwise} \end{cases}$$

The big-step operational semantics uses judgements of the form $\sigma, C \Downarrow_\gamma o : \sigma'$, read: given implementation context γ and starting from state σ , the execution of command C results in outcome $o \in O = \{ok, err, miss\}$ and state σ' . The outcome o can either equal: *ok* (elided to avoid clutter where possible), meaning that the execution was successful; *err*, meaning that the execution faulted with a language error, or *miss*, meaning that the execution faulted with a missing resource error.

Definition 3.5 (Operational Semantics). The big-step operational semantics is given in Figure 1 (all success cases), and Figure 2 (representative error cases).

The successful cases of the semantics are straightforward: for example, the nondet command generates an arbitrary natural number; the function call executes the function body in a store where the function parameters are given the values of the arguments of the function call and the local variables of the function are initialised to null; and the control flow statements behave as expected. Allocation requires the specified amount of contiguous cells (always available as heaps are finite), and lookup, mutation, and deallocation require the targeted cell not to have been freed.

The semantics stores error information in a dedicated program variable *err*, which is not available to the programmer. For simplicity of error messages, we assume to have a function $\text{str} : \text{PExp} \rightarrow \text{Str}$, which serialises program expressions into strings. The error cases of the semantics are split into language errors, which can be captured by program-logic reasoning, and missing resource errors, which cannot. Language errors arise due to, for example: expressions evaluating to \downarrow because a variable is missing from the store, sub-expressions being incorrectly typed (e.g. $\text{null} + 1$), or operators being partial (e.g. $0 - 5$ landing out of Nat); the access of deallocated cells (i.e., the use-after-free error); incorrect typing in commands (e.g., non-Booleans in the condition of the *if* or *while* statements); and the calling of non-existent functions. On the other hand, missing resource errors arise from accessing cells that are not present in memory.

⁶OX/UX frame preservation essentially means that if a program successfully runs from/to a given initial/final state, then it will also successfully run from/to an extended initial/final state, and that extension (referred to as frame) will not be affected by the execution. It is known that losing deallocation information breaks UX frame preservation, because then it would be possible to frame on the deallocated cells onto the final state, but not the initial state [40]. The solution is to keep explicit track of deallocated cells, which we achieve through the use of \emptyset .

$$\begin{array}{c}
\frac{}{\sigma, \text{skip} \Downarrow_Y \sigma} \quad \frac{\llbracket E \rrbracket_s = v \quad s' = s[x \rightarrow v]}{(s, h), x := E \Downarrow_Y (s', h)} \quad \frac{n \in \mathbb{N} \quad s' = s[x \rightarrow n]}{(s, h), x := \text{nondet} \Downarrow_Y (s', h)} \quad \frac{\llbracket E \rrbracket_s = \text{true} \quad (s, h), C_1 \Downarrow_Y \sigma'}{(s, h), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_Y \sigma'} \\
\\
\frac{\llbracket E \rrbracket_s = \text{false} \quad (s, h), C_2 \Downarrow_Y \sigma'}{(s, h), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_Y \sigma'} \quad \frac{\llbracket E \rrbracket_s = \text{false}}{(s, h), \text{while } (E) C \Downarrow_Y (s, h)} \quad \frac{\llbracket E \rrbracket_s = \text{true} \quad (s, h), C \Downarrow_Y \sigma' \quad \sigma'', \text{while } (E) C \Downarrow_Y \sigma'}{(s, h), \text{while } (E) C \Downarrow_Y \sigma'} \\
\\
\frac{\sigma, C_1 \Downarrow_Y \sigma'' \quad \sigma'', C_2 \Downarrow_Y \sigma'}{\sigma, C_1; C_2 \Downarrow_Y \sigma'} \quad \frac{f(\vec{x}) \{C; \text{return } E'\} \in \gamma \quad \llbracket \tilde{E} \rrbracket_s = \vec{v} \quad \text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\} \quad s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \quad (s_p, h), C \Downarrow_Y (s_q, h') \quad \llbracket E' \rrbracket_{s_q} = v'}{(s, h), y := f(\tilde{E}) \Downarrow_Y (s[y \rightarrow v'], h')} \quad \frac{\llbracket E \rrbracket_s = n \quad h(n) = v}{(s, h), x := [E] \Downarrow_Y (s[x \rightarrow v], h)} \\
\\
\frac{\llbracket E_1 \rrbracket_s = n \quad h(n) \in \text{Val} \quad \llbracket E_2 \rrbracket_s = v \quad h' = h[n \mapsto v]}{(s, h), [E_1] := E_2 \Downarrow_Y (s, h')} \quad \frac{\llbracket E \rrbracket_s = n \quad (n' + i \notin \text{dom}(h))_{0 \leq i < n} \quad h' = h[n' \mapsto \text{null}] \cdots [n' + n - 1 \mapsto \text{null}]}{(s, h), x := \text{new}(E) \Downarrow_Y (s[x \rightarrow n'], h')} \quad \frac{\llbracket E \rrbracket_s = n \quad h(n) \in \text{Val}}{(s, h), \text{free}(E) \Downarrow_Y (s, h[n \mapsto \emptyset])}
\end{array}$$

Fig. 1. Operational semantics, successful cases

$$\begin{array}{c}
\frac{\llbracket E_1 \rrbracket_s = \zeta \quad v_{\text{err}} = [\text{"ExprEval"}, \text{str}(E_1)]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{err} : (s_{\text{err}}, h)} \quad \frac{\llbracket E_1 \rrbracket_s = n \notin \text{dom}(h) \quad v_{\text{err}} = [\text{"MissingCell"}, \text{str}(E_1), n]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{miss} : (s_{\text{err}}, h)} \quad \frac{\llbracket E_1 \rrbracket_s = n \quad h(n) = \emptyset \quad v_{\text{err}} = [\text{"UseAfterFree"}, \text{str}(E_1), n]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{err} : (s_{\text{err}}, h)} \\
\\
\frac{\llbracket E \rrbracket_s = v \quad v_{\text{err}} = [\text{"Error"}, v]}{(s, h), \text{error}(E) \Downarrow_Y \text{err} : (s_{\text{err}}, h)} \quad \frac{\sigma, C_1 \Downarrow_Y \text{err} : \sigma'}{\sigma, C_1; C_2 \Downarrow_Y \text{err} : \sigma'} \quad \frac{\llbracket E \rrbracket_s = \text{true} \quad (s, h), C \Downarrow_Y \text{err} : (s, h)}{(s, h), \text{while } (E) C \Downarrow_Y \text{err} : (s, h)}
\end{array}$$

Fig. 2. Operational semantics, faulting cases (excerpt), with $s_{\text{err}} \triangleq s[\text{err} \rightarrow v_{\text{err}}]$ and $\text{str} : \text{PExp} \rightarrow \text{Str}$.

4 EXACT SEPARATION LOGIC

We introduce an exact separation logic for our programming language, giving the assertion language in §4.1, specifications in §4.2, and the program logic rules in §4.3.

4.1 Assertion Language

To define assertions and their meaning, we introduce logical variables, $x, y, z, \in \text{LVar}$, distinct from program variables, and define the set of logical expressions as follows:

$$E \in \text{LExp} \triangleq v \mid x \mid x \mid E + E \mid E - E \mid \dots \mid E = E \mid \neg E \mid E \wedge E \mid \dots \mid E \cdot E \mid E : E \mid \dots$$

Note that we can use program expressions in assertions (for example, $E \in \text{Val}$), as they form a proper subset of logical expressions.

Definition 4.1 (Assertion Language). The assertion language is defined as follows:

$$\pi \in \text{BASrt} \triangleq E_1 = E_2 \mid E_1 < E_2 \mid E \in X \mid \dots \mid \neg \pi \mid \pi_1 \Rightarrow \pi_2$$

$$P \in \text{Asrt} \triangleq \pi \mid \text{False} \mid P_1 \Rightarrow P_2 \mid \exists x. P \mid \text{emp} \mid E_1 \mapsto E_2 \mid E \mapsto \emptyset \mid P_1 \star P_2 \mid \bigotimes_{E_1 \leq x < E_2} P$$

where $E, E_1, E_2 \in \text{LExp}$, $X \subseteq \text{Val}$, and $x \in \text{LVar}$.

Boolean assertions, $\pi \in \text{BASrt}$, lift Boolean logical expressions to assertions. Assertions, $P \in \text{Asrt}$, contain Boolean assertions, standard first-order connectives and quantifiers, and spatial assertions. Spatial assertions include the empty memory assertion, emp , the positive cell assertion $E_1 \mapsto E_2$, and the negative cell assertion $E \mapsto \emptyset$ as found in [17–19, 21], and in ISL as $E \not\mapsto$ [40], and assertions built from separating conjunction (star) and its iteration (iterated star).

To define assertion satisfiability, we introduce substitutions, $\theta : \text{LVar} \rightarrow_{\text{fin}} \text{Val}$, which are partial finite mappings from logical variables to values, extending expression evaluation of Definition 3.4 to $\llbracket E \rrbracket_{\theta,s}$ straightforwardly, with a new base case for logical variables:

$$\llbracket x \rrbracket_{E,s} = \theta(x), \text{ if } x \in \text{dom}(\theta) \quad \llbracket x \rrbracket_{\theta,s} = \text{?}, \text{ if } x \notin \text{dom}(\theta)$$

Definition 4.2 (Satisfiability). The satisfiability relation is first defined for Boolean assertions, denoted by $\theta, s \models \pi$, and is then lifted to arbitrary assertions, denoted by $\theta, \sigma \models P$, as follows:

$$\begin{array}{ll} \theta, (s, h) \models & \\ \theta, s \models & \pi \quad \Leftrightarrow \quad \theta, s \models \pi \wedge h = \emptyset \\ & \text{False} \quad \Leftrightarrow \quad \text{never} \\ E_1 = E_2 & \Leftrightarrow \quad \llbracket E_1 = E_2 \rrbracket_{\theta,s} = \text{true} \quad P_1 \Rightarrow P_2 \quad \Leftrightarrow \quad \theta, (s, h) \models P_1 \Rightarrow \theta, (s, h) \models P_2 \\ E_1 < E_2 & \Leftrightarrow \quad \llbracket E_1 < E_2 \rrbracket_{\theta,s} = \text{true} \quad \exists x. P \quad \Leftrightarrow \quad \exists v \in \text{Val}. \theta[x \mapsto v], (s, h) \models P \\ E \in X & \Leftrightarrow \quad \llbracket E \rrbracket_{\theta,s} \in X \quad \text{emp} \quad \Leftrightarrow \quad h = \emptyset \\ \pi_1 \Rightarrow \pi_2 & \Leftrightarrow \quad \theta, s \models \pi_1 \Rightarrow \theta, s \models \pi_2 \quad E_1 \mapsto E_2 \quad \Leftrightarrow \quad h = \{\llbracket E_1 \rrbracket_{\theta,s} \mapsto \llbracket E_2 \rrbracket_{\theta,s}\} \\ \neg(E_1 = E_2) & \Leftrightarrow \quad \llbracket E_1 = E_2 \rrbracket_{\theta,s} = \text{false} \quad E_1 \mapsto \emptyset \quad \Leftrightarrow \quad h = \{\llbracket E_1 \rrbracket_{\theta,s} \mapsto \emptyset\} \\ \neg(E_1 < E_2) & \Leftrightarrow \quad \llbracket E_1 < E_2 \rrbracket_{\theta,s} = \text{false} \quad P_1 \star P_2 \quad \Leftrightarrow \quad \exists h_1, h_2. h = h_1 \uplus h_2 \wedge \\ & \quad \theta, (s, h_1) \models P_1 \wedge \theta, (s, h_2) \models P_2 \\ \neg(E \in X) & \Leftrightarrow \quad \llbracket E \rrbracket_{\theta,s} \notin X \quad \textcircled{\star}_{E_1 \leq x < E_2} P \quad \Leftrightarrow \quad (i < k \wedge \exists h_i, \dots, h_{k-1}. h = \uplus_{j=i}^{k-1} h_j \wedge \\ & \quad \forall j. i \leq j < k \Rightarrow \theta, (s, h_j) \models P[j/x]) \vee \\ \neg(\pi_1 \Rightarrow \pi_2) & \Leftrightarrow \quad \theta, s \models \pi_1 \wedge \theta, s \models \neg \pi_2 \quad (i \geq k \wedge h = \emptyset), \text{ where } i = \llbracket E_1 \rrbracket_{\theta,s}, k = \llbracket E_2 \rrbracket_{\theta,s} \\ \neg \neg \pi & \Leftrightarrow \quad \theta, s \models \pi \quad \text{and } x \text{ is not featured in either } E_1 \text{ or } E_2. \end{array}$$

Note that Boolean assertion satisfiability (expectedly) does not depend on the heap, but also that due to Boolean expression evaluation being three-valued (true, false, or ?), negation has to be defined case-by-case for Boolean assertions, rather than using the negation of the meta-logic. The satisfiability cases for first-order and spatial assertions are defined in the standard way. For convenience, we choose Boolean assertions to be satisfiable only in the empty heap. Also, note that the iterated star defaults to emp if the upper bound is not greater than the lower.

Definition 4.3 (Validity and Entailment). An assertion P is valid, denoted by $\models P$, iff $\forall \theta, \sigma. \theta, \sigma \models P$. An assertion P entails an assertion Q , denoted by $P \models Q$, iff $\forall \theta, \sigma. \theta, \sigma \models P \Rightarrow \theta, \sigma \models Q$.

4.2 Specifications

We define specifications for commands and functions, focussing in particular on external and internal function specifications and the relationship between them, as well as various forms of specification validity.

Definition 4.4. Specifications, $t = (P) (ok : Q_{ok}) (err : Q_{err}) \in \text{Spec} : \text{Asrt} \times \text{Asrt} \times \text{Asrt}$, comprise a pre-condition, P , a success post-condition, Q_{ok} , and the faulting post-condition, Q_{err} .

We denote that command C has specification t by $C : t$, or $(P) C (ok : Q_{ok}) (err : Q_{err})$ in quadruple form. Additionally, we use the following shorthand:

$$\begin{aligned} (P) C (Q) &\triangleq (P) C (ok : Q) (err : \text{False}) \\ (P) C (err : Q) &\triangleq (P) C (ok : \text{False}) (err : Q) \\ (P) C (Q) &\triangleq (P) C (ok : -) (err : -) \end{aligned}$$

noting the use of the calligraphic Q for cases in which the post-condition details are not relevant.

The validity of a specification t for a command C in an implementation context γ requires both OX and UX frame-preserving validity.

Definition 4.5 (γ -Valid Specifications). Given implementation context γ , command C , and specification $t = (P) (ok : Q_{ok}) (err : Q_{err})$, the specification t of command C is γ -valid, if and only if:

// Frame-preserving over-approximating validity
 $(\forall \theta, s, h, h_f, o, s', h''. \theta, (s, h) \models P \implies$
 $(s, h \uplus h_f), C \Downarrow_{\gamma} o : (s', h'') \implies (o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, (s', h') \models Q_o)) \wedge$
// Frame-preserving under-approximating validity
 $(\forall \theta, s', h', h_f, o, \theta, (s', h') \models Q_o \implies h_f \nVdash h' \implies$
 $(\exists s, h. \theta, (s, h) \models P \wedge (s, h \uplus h_f), C \Downarrow_{\gamma} o : (s', h' \uplus h_f)))$

We write $\gamma \models C : t$ or $\gamma \models (P) C (ok : Q_{ok}) (err : Q_{err})$ to denote that a specification $t = (P) C (ok : Q_{ok}) (err : Q_{err})$ of command C is γ -valid.

Observe that the outcome o can either be success or a language error; it cannot be a missing resource error as this would break frame preservation. As our operational semantics is complete, we can also use ESL to characterise non-termination. In particular, if a command satisfies a specification in which both post-conditions are False, then it is guaranteed to not terminate if executed from a state satisfying the pre-condition. Were the semantics incomplete (e.g., if it did not reason about errors), then such a specification might also indicate the absence of a semantic transition.

Functions have two kinds of specifications: external specifications, which provide the interface the function exposes to the client, and the related internal specifications, which provide the interface to the function implementation. This terminology is also used informally in InsecSL [35].

Definition 4.6 (External Specifications). A specification $(P) (ok : Q_{ok}) (err : Q_{err})$ is an external function specification (also: external specification) if and only if

- $P = (\vec{x} = \vec{x} \star P')$, for some distinct program variables \vec{x} , distinct logical variables \vec{x} , and assertion P' , with $\text{pv}(P') = \emptyset$
- $\text{pv}(Q_{ok}) = \{\text{ret}\}$ or $Q_{ok} = \text{False}$
- $\text{pv}(Q_{err}) = \{\text{err}\}$ or $Q_{err} = \text{False}$

The set of external specifications is denoted by \mathcal{ESpec} .

Definition 4.7 (Function Specification Contexts). A function specification context (also: specification context), Γ , is a finite partial function from function identifiers to a set of external specifications:

$$\Gamma \in \text{Fid} \rightarrow_{\text{fin}} \mathcal{P}(\mathcal{ESpec})$$

The constraints on the program variables in external specifications are well-known from OX logics and follow the usual scoping of the parameters and local variables of functions: the pre-conditions only contain the function parameters, \vec{x} ; and the post-conditions only have the (dedicated) program variables ret or err , which hold, respectively, the return value of the function on successful termination or the error value on faulting termination. No other program variables can be present in the two post-conditions due to variable scope being limited to the function body.

Internal function specifications are more interesting for exact and UX reasoning. The internal pre-condition is straightforward, simply extending the external pre-condition by instantiating the locals to null. The internal post-condition must include information about the parameters and local variables, as no information can be lost from the pre- to the post-condition. This means that the connection between internal and external specifications is subtle, given the constraints on external post-conditions. To address this, we define an internalisation function, relating an external function specification with a set of possible internal specifications. In particular, the external post-condition is required to be equivalent to the internal one in which the parameters and local variables of the internal post-condition have been replaced by fresh existentially quantified logical variables.

Definition 4.8 (Internalisation). Given implementation context γ and function $f \in \text{dom}(\gamma)$, a function specification internalisation, $\text{Int}_{\gamma,f} : \mathcal{ESpec} \rightarrow \mathcal{P}(\text{Spec})$ is defined as follows:

$$\begin{aligned} \text{Int}_{\gamma,f}((P) (ok : Q_{ok}) (err : Q_{err})) = \\ \{(P \star \bar{z} = \text{null}) (ok : Q'_{ok}) (err : Q'_{err}) \mid \begin{aligned} &\models Q'_{ok} \Rightarrow E \in \text{Val} \text{ and} \\ &\models Q_{ok} \Leftrightarrow \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \text{ret} = E[\vec{p}/\vec{p}] \text{ and} \\ &\models Q_{err} \Leftrightarrow \exists \vec{p}. Q'_{err}[\vec{p}/\vec{p}]\}, \end{aligned}$$

where $f(\vec{x})\{C; \text{return } E\} \in \gamma$, $\bar{z} = \text{pv}(C) \setminus \text{pv}(P)$, $\vec{p} = \text{pv}(P) \uplus \{\bar{z}\}$, and the logical variables \vec{p} are fresh with respect to Q_{ok} and Q_{err} .

This approach works for OX logics as well. It is not necessary, however, as information about program variables can be forgotten in internal post-conditions using forward consequence, making the internal post-conditions simpler.

Definition 4.9 (Environments). An environment, (γ, Γ) , is a pair consisting of an implementation context γ and a specification context Γ .

An environment (γ, Γ) is valid if and only if every function specified in Γ has an implementation in γ and every specification in Γ has a γ -valid internal specification.

Definition 4.10 (Valid Environments). Given an implementation context γ and a specification context Γ , the environment (γ, Γ) is valid, written $\models (\gamma, \Gamma)$, if and only if

$$\begin{aligned} \text{dom}(\Gamma) \subseteq \text{dom}(\gamma) \wedge \\ (\forall f, \vec{x}, C, E. f(\vec{x})\{C; \text{return } E\} \in \gamma \implies (\forall t. t \in \Gamma(f) \implies \exists t' \in \text{Int}_{\gamma,f}(t). \gamma \models C : t')) \end{aligned}$$

Finally, a specification t is valid for a command C in a specification context Γ if and only if t is γ -valid for all implementation contexts γ that validate Γ .

Definition 4.11 (Γ -Valid Specifications). Given a specification context Γ , a command C , and a specification $t = (P) (Q)$, the specification t is valid for command C in Γ (also: Γ -valid), written $\Gamma \models C : t$ or $\Gamma \models (P) C (Q)$, if and only if:

$$\forall \gamma. \models (\gamma, \Gamma) \implies \gamma \models (P) C (Q)$$

4.3 Program Logic

The rules for the program logic are given in Figure 3 for basic commands, Figure 4 for composite commands and structural rules, and Figure 5 for the function-related rules. In the figures, we only give an excerpt of error-related rules (all are available in Appendix B) and denote the repetition of the pre-condition in the post-condition by *pre*. When reading these rules, it is important to remember that the judgements must not lose information and must cover all paths. The judgement $\Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err})$ means that the specification t is derivable for a command C given the specifications recorded in Γ , whereas the judgement $\vdash (\gamma, \Gamma)$ means that the environment (γ, Γ) is well-formed, i.e. constructed through the [ENV-EMPTY] and [ENV-EXTEND] rules.

The basic command rules are fairly straightforward. The [NONDET] rule existentially quantifies the generated value (i.e., $x \in \mathbb{N}$) to capture all paths, in contrast with the rules featured in RHL [13] and ISL [40] which record an explicitly chosen value to describe one path. The $E' \in \text{Val}$ is necessary in the post-condition as we know that E' evaluates to a value from $x = E'$ in the pre-condition and exact reasoning cannot lose information; this also applies to a number of other rules. The [ASSIGN] rule requires that the program expression E evaluates to a value in the pre-condition ($E \in \text{Val}$), as we are working in an untyped setting. Strictly speaking, we should have an additional case in which the variable being assigned to is not in the store. For the logical reasoning, to avoid clutter, we instead

$$\begin{array}{c}
\text{NONDET} \quad \frac{x \notin \text{pv}(E') \quad Q \triangleq E' \in \text{Val} \star x \in \mathbb{N}}{\Gamma \vdash (x = E') \ x := \text{nondet} \ (Q)} \quad \text{ASSIGN} \quad \frac{x \notin \text{pv}(E') \quad Q \triangleq E' \in \text{Val} \star x = E[E'/x]}{\Gamma \vdash (x = E' \star E \in \text{Val}) \ x := E \ (Q)} \\
\\
\text{SKIP} \quad \Gamma \vdash (\text{emp}) \ \text{skip} \ (\text{emp}) \\
\\
\text{LOOKUP} \quad \frac{x \notin \text{pv}(E') \quad Q \triangleq E' \in \text{Val} \star x = E_1[E'/x] \star E[E'/x] \mapsto E_1[E'/x]}{\Gamma \vdash (x = E' \star E \mapsto E_1) \ x := [E] \ (Q)} \quad \text{MUTATE} \quad \frac{Q \triangleq E_1 \mapsto E_2 \star E \in \text{Val}}{\Gamma \vdash (E_1 \mapsto E \star E_2 \in \text{Val}) \ [E_1] := E_2 \ (Q)} \\
\\
\text{NEW} \quad \frac{x \notin \text{pv}(E') \quad Q \triangleq E' \in \text{Val} \star (\bigotimes_{0 \leq i < E[E'/x]} ((x+i) \mapsto \text{null}))}{\Gamma \vdash (x = E' \star E \in \mathbb{N}) \ x := \text{new}(E) \ (ok : Q)} \quad \text{FREE} \quad \frac{Q \triangleq E' \in \text{Val} \star E \mapsto \emptyset}{\Gamma \vdash (E \mapsto E') \ \text{free}(E) \ (ok : Q)} \\
\\
\text{ERROR} \quad \frac{E_{\text{err}} \triangleq [\text{"Error"}, E]}{\Gamma \vdash (E \in \text{Val}) \ \text{error}(E) \ (err : err = E_{\text{err}})} \quad \text{LOOKUP-ERR-VAL} \quad \frac{P \triangleq x = E' \star E \notin \text{Val} \quad E_{\text{err}} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (P) \ x := [E] \ (err : Q_{\text{err}})} \quad \text{LOOKUP-ERR-USE-AFTER-FREE} \quad \frac{P \triangleq x = E' \star E \mapsto \emptyset \quad E_{\text{err}} \triangleq [\text{"UseAfterFree"}, \text{str}(E), E]}{\Gamma \vdash (P) \ x := [E] \ (err : Q_{\text{err}})}
\end{array}$$

Fig. 3. ESL basic command rules (excerpt), with $Q_{\text{err}} = (\text{pre} \star \text{err} = E_{\text{err}})$

$$\begin{array}{c}
\text{IF-THEN} \quad \frac{C \triangleq \text{if} (E) \ C_1 \ \text{else} \ C_2 \quad \Gamma \vdash (P \wedge E) \ C_1 \ (Q)}{\Gamma \vdash (P \wedge E) \ C \ (Q)} \quad \text{IF-ELSE} \quad \frac{C \triangleq \text{if} (E) \ C_1 \ \text{else} \ C_2 \quad \Gamma \vdash (P \wedge \neg E) \ C_2 \ (Q)}{\Gamma \vdash (P \wedge \neg E) \ C \ (Q)} \quad \text{IF-ERR-VAL} \quad \frac{C \triangleq \text{if} (E) \ C_1 \ \text{else} \ C_2 \quad E_{\text{err}} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (P \star E \notin \text{Val}) \ C \ (err : Q_{\text{err}})} \\
\\
\text{SEQ} \quad \frac{\Gamma \vdash (P) \ C_1 \ (ok : R) \ (err : Q_{\text{err}}^1) \quad \Gamma \vdash (R) \ C_2 \ (ok : Q_{ok}) \ (err : Q_{\text{err}}^2)}{\Gamma \vdash (P) \ C_1; C_2 \ (ok : Q_{ok}) \ (err : Q_{\text{err}}^1 \vee Q_{\text{err}}^2)} \quad \text{WHILE-ITERATE} \quad \frac{\forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B} \quad \forall i \in \mathbb{N}. \Gamma \vdash (P_i \wedge E) \ C \ (ok : P_{i+1}) \ (err : Q_i)}{\Gamma \vdash (P_0) \ \text{while} \ (E) \ C \ (ok : \exists m. P_m) \ (err : \exists m. Q_m)} \\
\\
\text{EQUIV} \quad \frac{\Gamma \vdash (P') \ C \ (ok : Q'_{ok}) \ (err : Q'_{err}) \quad \models P', Q'_{ok}, Q'_{err} \Leftrightarrow P, Q_{ok}, Q_{err}}{\Gamma \vdash (P) \ C \ (ok : Q_{ok}) \ (err : Q_{err})} \quad \text{FRAME} \quad \frac{\text{mod}(C) \cap \text{fv}(R) = \emptyset \quad \Gamma \vdash (P) \ C \ (ok : Q_{ok}) \ (err : Q_{err})}{\Gamma \vdash (P \star R) \ C \ (ok : Q_{ok} \star R) \ (err : Q_{err} \star R)} \\
\\
\text{EXISTS} \quad \frac{\Gamma \vdash (P) \ C \ (ok : Q_{ok}) \ (err : Q_{err})}{\Gamma \vdash (\exists x. P) \ C \ (ok : \exists x. Q_{ok}) \ (err : \exists x. Q_{err})} \quad \text{DISJ} \quad \frac{\Gamma \vdash (P_1) \ C \ (ok : Q_{ok}^1) \ (err : Q_{\text{err}}^1) \quad \Gamma \vdash (P_2) \ C \ (ok : Q_{ok}^2) \ (err : Q_{\text{err}}^2)}{\Gamma \vdash (P_1 \vee P_2) \ C \ (ok : Q_{ok}^1 \vee Q_{ok}^2) \ (err : Q_{\text{err}}^1 \vee Q_{\text{err}}^2)}
\end{array}$$

Fig. 4. ESL composite-command and structural rules (excerpt)

assume that the program variables are always in the store because we are analysing function bodies and all local variables are initialised on function entry. The error-related rules capture cases in which expression evaluation faults (e.g. [LOOKUP-ERR-VAL] rule, using $E \notin \text{Val}$), expressions are of the incorrect type, or memory is accessed after it has been freed (e.g. [LOOKUP-ERR-USE-AFTER-FREE] rule, using $E \mapsto \emptyset$). Note that missing resource errors cannot be captured without breaking frame preservation, as the added-on frame could contain the missing resource.

When it comes to composite commands, we opt for two *if*-rules, covering the branches separately. The sequencing rule shows how exact quadruples of successive commands can be joined together, highlighting, in particular, how errors are collected using disjunction. The while rule is a simple adaptation of the RHL while rule [13]. Interestingly, it does not need adjustment for non-termination, as it can already capture it, since it is, in fact, a generalisation of the SL while rule.

$$\begin{array}{c}
\text{FCALL} \\
\frac{(\vec{x} = \vec{x} \star P) (Q_{ok})(Q_{err}) \in \Gamma(f) \quad y \notin \text{pv}(E_y)}{\Gamma \vdash (y = E_y \star \vec{E} = \vec{x} \star P) \ y := f(\vec{E}) \ (ok : \vec{E}[E_y/y] = \vec{x} \star Q_{ok}[y/\text{ret}]) \ (err : y = E_y \star \vec{E} = \vec{x} \star Q_{err})} \\
\\
\text{ENV-EMPTY} \\
\vdash (\emptyset, \emptyset) \\
\\
\text{ENV-EXTEND} \\
\vdash (\gamma, \Gamma) \quad I = \{1, \dots, n\} \quad \forall i \in I. f_i \notin \text{dom}(\gamma) \quad \gamma' = \gamma[f_i \mapsto (\vec{x}_i, C_i, E_i)]_{i \in I} \\
\Gamma(\alpha) = \Gamma[f_i \mapsto \{(P^i(\beta)) (ok : Q_{ok}^i(\beta)) (err : Q_{err}^i(\beta)) \mid \beta < \alpha\} \cup \{(P_\infty^i(\beta)) (\text{False}) \mid \beta \leq \alpha\}]_{i \in I} \\
\forall i \in I, \alpha. \exists t \in \text{Int}_{\gamma', f_i}((P^i(\alpha)) (ok : Q_{ok}^i(\alpha)) (err : Q_{err}^i(\alpha))). \Gamma(\alpha) \vdash C_i : t \\
\forall i \in I, \alpha. \exists t \in \text{Int}_{\gamma', f_i}((P_\infty^i(\alpha)) (\text{False})). \Gamma(\alpha) \vdash C_i : t \\
P^i \triangleq \exists \alpha. P^i(\alpha) \star \alpha \in \mathcal{O} \quad P_\infty^i \triangleq \exists \alpha. P_\infty^i(\alpha) \star \alpha \in \mathcal{O} \\
Q_{ok}^i \triangleq \exists \alpha. Q_{ok}^i(\alpha) \star \alpha \in \mathcal{O} \quad Q_{err}^i \triangleq \exists \alpha. Q_{err}^i(\alpha) \star \alpha \in \mathcal{O} \\
\Gamma'' := \Gamma[f_i \mapsto \{(P^i) (ok : Q_{ok}^i) (err : Q_{err}^i), (P_\infty^i) (\text{False})\}]_{i \in I} \\
\hline
\vdash (\gamma', \Gamma'')
\end{array}$$

Fig. 5. ESL function-related rules (excerpt)

The structural rules are not surprising, with equivalence replacing the forward consequence of OX reasoning and backward consequence of UX reasoning, and with frame, existential elimination and disjunction affecting both post-conditions. Disjunction allows us to derive the standard SL if rule, which captures both branches at the same time. Note, however, the absence of a sound conjunction rule, due to the fact that the conjunction rules of SL and ISL cannot be combined in ESL, as conjunction does not distribute over the star in both directions, breaking frame preservation.

We discuss the function-related rules in detail, starting from [FCALL], whose premises are standard, but whose pre- and post-condition are adjusted for sound UX reasoning. In particular, in the pre-condition, the usual $P'[\vec{E}/\vec{x}]$ assertion (P' can have program variables in \vec{x}) now has the form $\vec{E} = \vec{x} \star P$ (where P has no program variables). This is required because the connection between the passed function arguments and the logical variables has to be maintained in the post-condition as well, with $\vec{E}[E_y/y] = \vec{x}$. Otherwise, the rule would not be UX-sound as it could lose information about the program variables of the calling function.

The [ENV-EXTEND] rule highlights our need for different treatment of terminating and non-terminating specifications. In particular, at each extension we add a cluster of mutually recursive functions $\{f_i \mid i \in \{1, \dots, n\}\}$, imposing a joint non-negative measure on the specifications, denoted by α in the set of computable ordinals $\mathcal{O} \triangleq \omega_1^{\text{CK}}$. Extending the measure beyond natural numbers allows us to reason about a broader set of functions, such as those with non-deterministic nested recursion. We require that any recursive function call of any added function may use a terminating specification of any f_i only if its measure is strictly smaller than α , or a non-terminating specification of any f_i with measure less or equal to α . If this distinction were not in place, that is, if we were to try to use the standard SL rule:

$$\frac{\vdash (\gamma, \Gamma) \quad I = \{1, \dots, n\} \quad \forall i \in I. f_i \notin \text{dom}(\gamma) \quad \gamma' = \gamma[f_i \mapsto (\vec{x}_i, C_i, E_i)]_{i \in I} \quad \Gamma' = \Gamma[f_i \mapsto \{t_i \mid i \in I\}] \quad \forall i \in I. \exists t \in \text{Int}_{\gamma', f_i}(t_i). \vdash C_i : t}{\vdash (\gamma', \Gamma')}$$

then we would be able to prove unsound specifications of non-terminating functions. For example, we would be able to prove that the non-terminating function $f() \{ r := f(); \text{return } r \}$ satisfies the sound specification (emp) $f() (\text{False})$, but also the unsound specification (emp) $f() (\text{ret} = 42)$. This is not an issue in OX logics because the meaning of triples is conditional on function termination.

In UX logics, it would imply the existence of an execution path from the pre- to the post-condition, contradicting the non-termination of f . In the end, the measure α is abstracted into a logical variable in the final specification added to Γ'' , and can normally be eliminated using equivalence, as shown in examples in §5. Finally, note that, for each f_i , we provide one terminating and one non-terminating specification. We can generalise to an arbitrary number of specifications, but this would complicate the presentation without introducing additional ideas.

4.4 Soundness

We state the soundness results for ESL and give intuition about the proofs, which can be found integrally in Appendices C, D, and E.

THEOREM 4.12. *Any derivable specification is valid:*

$$\forall \Gamma, P, C, Q. \Gamma \vdash (P) C (Q) \implies \Gamma \models (P) C (Q)$$

PROOF. By induction on $\Gamma \vdash (P) C (Q)$. Most cases are straightforward; the [FCALL] rule obtains a valid specification for the function body from the validity of the environment. \square

THEOREM 4.13. *Any well-formed environment is valid:*

$$\forall \gamma, \Gamma. \vdash (\gamma, \Gamma) \implies \models (\gamma, \Gamma)$$

When a specification can be used to prove itself (e.g. any specification in SL, and the non-terminating (NT) specifications in ESL), a form of fixpoint induction, called Scott induction [45], is required, which we use with three slightly varying instantiations to prove Theorem 4.13. We give the outline of the proof below.

PROOF. At the core of the proof is a lemma that states that $\models (\gamma, \Gamma) \implies (\forall \alpha. \models (\gamma', \Gamma(\alpha)))$, where γ' and $\Gamma(\alpha)$ have been obtained from γ and Γ as in the [ENV-EXTEND] rule. Using this lemma, and also showing that existential elimination can be soundly lifted to function specifications, we derive the desired $\models (\gamma', \Gamma'')$, where Γ'' is obtained from Γ and $\Gamma(\alpha)$ as in the [ENV-EXTEND] rule.

The proof of this lemma is done by transfinite induction on α , which has standard zero, successor, and limit ordinal cases. For clarity, we outline the proof for the case in which a single function f with body C_f is added; the generalisation to n mutually recursive functions is straightforward.

In all three induction cases, the soundness of all specifications except the non-terminating (NT) specification with the highest considered ordinal follows straightforwardly from the inductive hypothesis. This remaining NT-specification is vacuously UX-valid, meaning that we only need to prove its OX-validity, for which we use Scott induction [45].

We set up the Scott induction by extending the set of commands with two pseudo-commands, scope and choice, with the former modelling the function call but allowing arbitrary commands to be executed in place of the function body, and the latter denoting non-deterministic choice.

We then construct the greatest-fixpoint closure of these extended commands, denoted by \mathbb{C} , whose elements may contain infinite applications of the command constructors. We define a behavioural equivalence relation $\simeq_{\gamma'}$ on \mathbb{C} and denote by $\mathbb{C}_{\gamma'}$ the obtained quotient space. This relation induces a partial order $\sqsubseteq_{\gamma'}$, and a join operator that coincides with choice, and we show that $(\mathbb{C}_{\gamma'}, \sqsubseteq_{\gamma'})$ is a domain.

We next define S^α as the set of all equivalence classes that hold an element that, for every specification in $(\Gamma(\alpha))(f)$, OX-satisfies at least one of its internal specifications, and show that S^α is an admissible subset of $\mathbb{C}_{\gamma'}$, that is, that it contains the least element of $\mathbb{C}_{\gamma'}$ (represented, for example, by the infinite loop `while (true) { skip }`) and is chain-closed.

We then define the function $h(C) \triangleq C_f[C, \gamma', f]$, which replaces all function calls to f in C_f with C using the scope command, and the function g as the lifting of h to \mathbb{C}_γ : $g([C]) := [h(C)]$.

We next prove that g is continuous (that is, monotonic and supremum-preserving) and that $g(S^\alpha) \subseteq S^\alpha$, from which we can apply the Scott induction principle, together with a well-known identity of the least-fixpoint, which implies that $C_f \in \text{lfp}(g)$, to obtain that $[C_f] \in S^\alpha$. From there, we are finally able to prove that $\models (\gamma', \Gamma(\alpha))$. \square

These two theorems, to the best of our knowledge, are the first to demonstrate sound functional compositionality for non-OX logics. In particular, the proof of Theorem 4.13 can be adjusted for ISL (for which the function call rule is the same, but the [ENV-EXTEND] rule does not include non-terminating specifications) by removing the part using Scott induction. On the other hand, the Scott induction itself can be easily adapted for SL. We were not able to find an SL-proof in the literature, and wonder whether our proof is the first complete proof of its kind.

Admissible Properties of Function Specifications. We conclude by discussing how the structural rules of the logic transfer to specifications. In particular, given a valid function specification:

$$(P) f(\vec{x}) \text{ (ok : } Q_{ok} \text{) (err : } Q_{err} \text{)}$$

the specifications obtained from it using equivalence, frame, and existential introduction:

$$\begin{aligned} & (P') f(\vec{x}) \text{ (ok : } Q'_{ok} \text{) (err : } Q'_{err} \text{)} \\ & (P \star R) f(\vec{x}) \text{ (ok : } Q_{ok} \star R \text{) (err : } Q_{err} \star R \text{)} \\ & (\exists x. P) f(\vec{x}) \text{ (ok : } \exists x. Q_{ok} \text{) (err : } \exists x. Q_{err} \text{)} \end{aligned}$$

where P' , Q'_{ok} , and Q'_{err} are equivalent, respectively, to P , Q_{ok} , and Q_{err} , R does not contain program variables, and x is an arbitrary logical variable, are also valid.

5 EXAMPLES: LIST ALGORITHMS

We demonstrate how to use ESL to specify and verify correctness, incorrectness, and non-termination properties of recursive and iterative functions, using standard singly-linked list algorithms as demonstrator examples. In doing so, we give a number of observations from our specific ESL reasoning which are relevant to EX and UX reasoning in general. In particular, we focus on the difference between losing information via OX reasoning and hiding information via abstraction, highlighting strictly exact abstractions which play a fundamental role in compositional symbolic execution (cf. §6). Further examples, illustrating mutual recursion, can be found in Appendix F.

List Predicates. We implement singly-linked lists (onward: lists) in the standard way: every list node consists of two contiguous cells in the heap, with the first holding the value of the node, the second holding a pointer to the next node in the list, and the list terminating with a null pointer. To capture lists in ESL, we use several standard list predicates:

$$\begin{aligned} \text{list}(x) &\triangleq (x = \text{null}) \vee (\exists v, x'. x \mapsto v, x' \star \text{list}(x')) \\ \text{list}(x, n) &\triangleq (x = \text{null} \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star \text{list}(x', n - 1)) \\ \text{list}(x, \text{vs}) &\triangleq (x = \text{null} \star \text{vs} = []) \vee (\exists v, x', \text{vs}'. x \mapsto v, x' \star \text{list}(x', \text{vs}') \star \text{vs} = v : \text{vs}') \\ \text{list}(x, \text{xs}) &\triangleq (x = \text{null} \star \text{xs} = []) \vee (\exists v, x', \text{xs}'. x \mapsto v, x' \star \text{list}(x', \text{xs}') \star \text{xs} = x' : \text{xs}') \\ \text{list}(x, \text{xs}, \text{vs}) &\triangleq (x = \text{null} \star \text{xs} = [] \star \text{vs} = []) \vee \\ &\quad (\exists v, x', \text{xs}', \text{vs}'. x \mapsto v, x' \star \text{list}(x', \text{xs}', \text{vs}') \star \text{xs} = x' : \text{vs}' \star \text{vs} = v : \text{vs}') \end{aligned}$$

These predicates expose different parts of the list structure in their parameters, hiding the rest via existential quantification: the $\text{list}(x)$ predicate hides all information about the represented mathematical list, just declaring that there is a singly-linked list at address x ; the $\text{list}(x, n)$ predicate

hides the internal node addresses and values, exposing the list length via the parameter n ; the $\text{list}(x, xs)$ predicate hides information about the values of the mathematical list, exposing the internal addresses of the list via the parameter xs ; the $\text{list}(x, vs)$ predicate hides information about the internal addresses, exposing the list's values via the parameter vs ; and the strictly exact list predicate $\text{list}(x, xs, vs)$ hides nothing, exposing the entire node-value structure via the parameters xs and vs . In the following examples, we investigate, for the first time, the use of such predicates in non- OX program logics. These predicates are related to each other via logical equivalence as follows:

$$\begin{aligned} \text{list}(x) &\Leftrightarrow \exists n/vs/xs. \text{list}(x, n/vs/xs) \Leftrightarrow \exists xs, vs. \text{list}(x, xs, vs) \\ \text{list}(x, n) &\Leftrightarrow \exists vs/xs. \text{list}(x, vs/xs) \star |vs/xs| = n \Leftrightarrow \exists xs, vs. \text{list}(x, xs, vs) \star |xs/vs| = n \\ \text{list}(x, xs/vs) &\Leftrightarrow \exists vs/xs. \text{list}(x, xs, vs) \end{aligned}$$

List Length: Recursion, Iteration. We verify correctness of a recursive and an iterative implementation of the $\text{LLen}(x)$ function, which returns the length of a given list starting at address x . In doing so, we illustrate how to handle the measure for recursive function calls, how the folding of predicates works in the presence of equivalence, and how to move between external and internal specifications. The implementations are given in Figure 6 (left and middle), and the corresponding proof sketches are given in Figure 7. The specification we prove for both is standard:

$$(x = x \star \text{list}(x, n)) \text{ LLen}(x) (\text{list}(x, n) \star \text{ret} = n)$$

We first prove the recursive implementation, where we start by defining a decreasing measure on the pre-condition, which in this case is trivially n . Denoting the function body by C and using the $[\text{ENV-EXTEND}]$ rule, we assume to have a well-formed environment (γ, Γ) , such that $\text{LLen} \notin \text{dom}(\gamma)$, and define, using $P(\alpha) \triangleq x = x \star \text{list}(x, n) \star \alpha = n$ and $Q(\alpha) \triangleq \text{list}(x, n) \star \text{ret} = n \star \alpha = n$:

$$\gamma' \triangleq \gamma[\text{LLen} \mapsto (\{x\}, C, r)] \quad \Gamma(\alpha) \triangleq \Gamma[\text{LLen} \mapsto \{(P(\beta)) (Q(\beta)) \mid \beta < \alpha\}]$$

Then, we construct the proof sketch in Figure 7 (left), starting from the internal pre-condition of LLen and arriving at the internal post-condition $Q' \triangleq Q'_1 \vee Q'_2$. Interestingly, it is not possible to fold $\text{list}(x, n)$ back in Q'_2 because the existentially quantified x' is still held in program variable x , which can be forgotten in OX logics, but not in ESL/ISL due to equivalence/backward consequence. This observation can be formulated generally as follows:

(O1) if the analysed code accesses data-structure information that the used predicate hides, then it might not be possible to fold that predicate in an ESL/ISL proof.

<pre> LLen(x) { if (x = null) { r := 0 } else { x := [x + 1]; r := LLen(x); r := r + 1 }; return r } </pre>	<pre> LLen(x) { r := 0 while (x ≠ null) { x := [x + 1]; r := r + 1 }; return r } </pre>	<pre> LFree(x){ if (x = null) { r := null } else { y := x; x := [x + 1]; free(y); free(y + 1); r := LFree(x) }; return r } </pre>
---	---	---

Fig. 6. List algorithms: iterative list-length (left); recursive list-length (middle); recursive list-free (right)

In such cases, the folding happens in the transition from the internal to the external post-condition, which forgets all program variables:

$$\begin{aligned}
& \exists x_q, r_q. Q' [x_q/x] [r_q/r] \star \text{ret} = r [x_q/x] [r_q/r] \\
& \Leftrightarrow \alpha = n \star \text{ret} = n \star ((x = \text{null} \star n = 0) \vee (\exists x_q, r_q, v, x'. x_q = x' \star x \mapsto v, x' \star \text{list}(x', n-1) \star r_q = n)) \\
& \Leftrightarrow \alpha = n \star \text{ret} = n \star ((x = \text{null} \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star \text{list}(x', n-1))) \text{ [[can fold now]]} \\
& \Leftrightarrow \alpha = n \star \text{ret} = n \star \text{list}(x, n) \star (n = 0 \vee n > 0) \Leftrightarrow Q(\alpha)
\end{aligned}$$

This, according to the [ENV-EXTEND] rule, yields a final Γ'' which contains the specification $(\exists \alpha. P(\alpha)) \text{LLen}(x) (\exists \alpha. Q(\alpha))$, from which we obtain the desired specification using equivalence.

Observe, however, that if the list-length code called another function whose pre-condition required the $\text{list}(x, n)$ predicate folded while having $x = x'$, the proof could not continue. We discuss this ESL/ISL-specific issue further in the upcoming client examples.

We move to the iterative list length algorithm, eliding the measure as there is no recursion. To state the loop variant, we use the list-segment predicate, defined as follows:

$$\text{lseg}(x, y, n) \triangleq (x = y \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star \text{lseg}(x', y, n-1))$$

and to apply the [WHILE-ITERATE] rule, we define:

$$P_i \triangleq \begin{cases} \exists j. \text{lseg}(x, x, i) \star \text{list}(x, j) \star n = i + j \star r = i, & \text{if } i < n \\ \text{False} & \text{otherwise} \end{cases}$$

and via the proof sketch show that its premises hold. On exiting the loop, the negation of the loop condition collapses the existentials m and j . We obtain the given internal post-condition, from which we then move to the desired external post-condition, similarly to the recursive version.

For this proof, we also use three equivalence lemmas:

$$\begin{aligned}
L1 : & \models \text{lseg}(x, y, n+1) \Leftrightarrow \exists x', v. \text{lseg}(x, x', n) \star x' \mapsto v, y \\
L2 : & \models \text{list}(\text{null}, j) \Leftrightarrow j = 0 \\
L3 : & \models \text{lseg}(x, \text{null}, n) \Leftrightarrow \text{list}(x, n)
\end{aligned}$$

$ \begin{aligned} & \Gamma(\alpha) \vdash \\ & (x = x \star \text{list}(x, n) \star \alpha = n \star r = \text{null}) \\ & \text{if } (x = \text{null}) \{ \\ & \quad (x = x \star \text{list}(x, n) \star \alpha = n \star r = \text{null} \star x = \text{null}) \\ & \quad r := 0 \\ & \quad (Q'_1 : x = x \star \text{list}(x, n) \star \alpha = n \star x = \text{null} \star r = 0) \\ & \} \text{ else } \{ \\ & \quad (x = x \star \text{list}(x, n) \star \alpha = n \star r = \text{null} \star x \neq \text{null}) \\ & \quad \left(\begin{array}{c} \exists v, x'. x = x \star x \mapsto v, x' \star \text{list}(x', n-1) \star \\ \alpha = n \star r = \text{null} \end{array} \right) \\ & \quad x := [x+1]; \\ & \quad \left(\begin{array}{c} \exists v, x'. x = x' \star \text{list}(x', n-1) \star \alpha - 1 = n - 1 \star \\ x \mapsto v, x' \star r = \text{null} \end{array} \right) \\ & \quad \text{[[As } \alpha - 1 < \alpha, \text{ we can use the spec for } \alpha - 1 \text{]]} \\ & \quad r := \text{LLen}(x); \\ & \quad \left(\begin{array}{c} \exists v, x'. x = x' \star x \mapsto v, x' \star \text{list}(x', n-1) \star \\ \alpha = n \star r = n - 1 \end{array} \right) \\ & \quad r := r + 1 \\ & \quad \left(\begin{array}{c} Q'_2 : \exists v, x'. x = x' \star x \mapsto v, x' \star \text{list}(x', n-1) \star \\ \alpha = n \star r = n \end{array} \right) \\ & \} \\ & (Q' : Q'_1 \vee Q'_2) \end{aligned} $	$ \begin{aligned} & \Gamma \vdash (x = x \star \text{list}(x, n) \star r = \text{null}) \\ & r := 0 \\ & (x = x \star \text{list}(x, n) \star r = 0) \\ & (P_0) \\ & \text{while } (x \neq \text{null}) \{ \\ & \quad (P_i \star x \neq \text{null}) \\ & \quad \left(\begin{array}{c} \exists j, v, x'. \text{lseg}(x, x, i) \star x \mapsto v, x' \star \\ \text{list}(x', j-1) \star n = i + j \star r = i \end{array} \right) \\ & \quad x := [x+1]; \\ & \quad \left(\begin{array}{c} \exists j, x', v. \text{lseg}(x, x', i) \star x' \mapsto v, x \star \text{list}(x, j) \star \\ n = (i+1) + j \star r = i \end{array} \right) \\ & \quad \left(\begin{array}{c} \exists j. \text{lseg}(x, x, i+1) \star \text{list}(x, j) \star \\ n = (i+1) + j \star r = i \end{array} \right) \text{ [[L1]]} \\ & \quad r := r + 1 \\ & \quad (P_{i+1}) \\ & \} \\ & \left(\begin{array}{c} \exists m, j. \text{lseg}(x, x, m) \star \text{list}(x, j) \star n = m + j \star \\ r = m \star x = \text{null} \end{array} \right) \\ & \left(\text{lseg}(x, \text{null}, n) \star r = n \star x = \text{null} \right) \text{ [[L2 + equiv]]} \\ & \left((\text{list}(x, n) \star \text{ret} = n) [r/\text{ret}] \star x = \text{null} \right) \text{ [[L3]]} \end{aligned} $
--	---

Fig. 7. List length algorithm proof sketch: recursive (left) and iterative (right).

which state, respectively, that we can separate a non-empty list segment into its last element and the rest, that the length of an empty list is zero, and that a null-terminated list-segment is a list.

These two proofs lead us to the following observation:

(O2) in ESL/ISL specifications, just as in OX reasoning, information hidden via predicates in the pre-condition may also remain hidden in the post-condition,

highlighting that hiding information inside predicates does not always lead to over-approximation. In particular, for list length, from the UX point of view no information is lost as it was never there in the first place.

List Free: Deallocation. We next consider an implementation of the $\text{LFree}(x)$ function (Figure 6, right), which frees a given list at address x . Its OX specification is $\{\text{list}(x)\} \text{LFree}([x]) \{\text{ret} = \text{null}\}$, but it does not transfer to ESL/ISL because no resource from the pre-condition can be forgotten in the post-condition. Instead, we have to expose the internal pointers of the list using the $\text{list}(x, xs)$ predicate and then explicitly state that they have been freed in the post-condition:

$$(x = x \star \text{list}(x, xs)) \text{LFree}([x]) (\text{freed}(x : xs) \star \text{ret} = \text{null})$$

where the $\text{freed}(xs)$ predicate is defined as follows:

$$\text{freed}(xs) \triangleq (xs = [\text{null}]) \vee (\exists x', xs'. xs = x : xs' \star x \mapsto \emptyset \star x + 1 \mapsto \emptyset \star \text{freed}(xs'))$$

This specification, which has to make freed addresses explicit, yields the following observation:

(O3) ESL/ISL specifications may reveal implementation details.

We give the proof sketch in Appendix F, as it carries no additional insight w.r.t. that of $\text{LLen}(x)$.

List Reverse. We also consider the $\text{LRev}(x)$ function, which reverses a given list at address x , and prove that it satisfies the following, almost standard, specification:

$$(x = x \star \text{list}(x, vs)) \text{LRev}(x) (\text{list}(\text{ret}, vs^\dagger) \star R)$$

where vs^\dagger denotes the reverse of the mathematical list. The only difference with respect to its OX counterpart is in the additional $R \triangleq (|vs| = 0 \star x = \text{null}) \vee (|vs| > 0 \star x \in \mathbb{N})$ in the post-condition, which has to be there to maintain information about x known from the pre-condition. For space reasons, we give the proof sketch in Appendix F.

Client Code: Degrees of Abstraction, Non-Termination. We conclude the examples with two somewhat contrived, yet illustrative clients of the previously specified functions. The first example, given in Figure 8 (left), reverses the tail of a given non-empty list before re-attaching it and then calculating the list length. This proof sketch exhibits two problems. First, $\text{list}(x, vs)$ cannot be folded back for the call to list-length, as y holds an internal list pointer x' . The way to circumvent this is to move via equivalence to the strictly exact $\text{list}(x, xs, vs)$ predicate, which exposes the internal pointers and allows the folding. However, we then run into the second problem, which is that our specification of $\text{LLen}(x)$ works with $\text{list}(x, n)$, not $\text{list}(x, xs, vs)$. For that, the only solution is to re-prove $\text{LLen}(x)$ with the specification

$$(x = x \star \text{list}(x, xs, vs)) \text{LLen}(x) (\text{list}(x, xs, vs) \star \text{ret} = |vs|)$$

which can then be used for this client, reinforcing (O3). However, if we proved this less abstract LLen specification first, then we could derive the initial, more abstract one from it via equivalence. This brings us to the following observation:

(O4) admissible properties of function specifications allow the degree of abstraction to be adjusted.

$ \begin{aligned} &\Gamma \vdash (\text{list}(x, v : vs) \star y = \text{null} \star r = \text{null}) \\ &y := [x + 1]; \\ &(\exists x'. x \mapsto v, x' \star \text{list}(x', vs) \star y = x' \star r = \text{null}) \\ &y := \text{LRev}(y); \\ &(\exists x'. x \mapsto v, x' \star \text{list}(y, vs^\dagger) \star r = \text{null}) \\ &[x + 1] := y \\ &(\exists x'. x \mapsto v, x' \star \text{list}(y, vs^\dagger) \star y = x' \star r = \text{null}) \\ &[[\text{Problem: Cannot fold. Solution: Move to list}(x, xs, vs)]] \\ &(\exists xs, x'. x \mapsto v, x' \star \text{list}(y, xs, vs^\dagger) \star y = x' \star r = \text{null}) \\ &(\exists xs, x'. \text{list}(x, x' : xs, v : vs^\dagger) \star y = x' \star r = \text{null}) \\ &[[\text{Problem: Predicate not appropriate for LLen}]] \\ &r := \text{LLen}(x); \end{aligned} $	<pre> LClient(x) { l := LLen(x); if (l < 5) { r := LFree(x); error("LTS") } else { if (l > 10) { while (true) { skip } } else { r := LRev(l) } }; return r } </pre>
---	---

Fig. 8. Example Clients

Interestingly, applying this observation to the list-free algorithm, we obtain the specification

$$(x = x \star \text{list}(x)) \text{ LFree}([x]) (\text{freed}([x]) \star \text{ret} = \text{null} \star (\exists xs. \text{freed}(xs)))$$

where the post-condition states that there exists some portion of memory that has been freed. This is as close as one can get to the OX list-free specification in exact and UX reasoning.

We note that specifications featuring only strictly exact predicates, such as the one for list-length given above, will play an important role in compositional symbolic execution for true bug-finding (cf. §6); all of our example algorithms can be easily specified in this style.

Our second client program is given in Figure 8 (right). It takes a list and: reverses it if its length is between 5 and 10; frees it and then throws a language List-Too-Short (LTS) error if its length is less than 5; and does not terminate otherwise. Its ESL specification is:

$$\begin{aligned}
&(x = x \star \text{list}(x, vs)) \\
&\text{LClient}(x) \\
&(ok : 5 \leq |vs| \leq 10 \star \text{list}(\text{ret}, vs^\dagger) \star R) (err : |vs| < 5 \star (\exists xs. \text{freed}(x : xs) \star |xs| = |vs|) \star err = \text{"LTS"})
\end{aligned}$$

where the assertion R is as given for list reverse; the proof sketch is given in Appendix F. The specification captures the successful and faulting behaviours explicitly, together with the conditions under which they occur, and carries two noteworthy points.

First, there is the question of which list predicate is appropriate for this client. As the list is being reversed in one branch, we believe that a useful predicate should contain node values. We chose $\text{list}(x, vs)$, but could have gone with $\text{list}(x, xs, vs)$ instead, obtaining a less abstract specification from which we could then derive the one given above by existentially quantifying xs .

Second, the non-terminating branch (when $|vs| > 10$) is implicit, in that it is subsumed by the success post-condition (since $P \vee (|vs| > 10 \star \text{False}) \Leftrightarrow P$). However, to demonstrate that it exists, we can constrain the pre-condition appropriately to prove the (partial) specification:

$$(x = x \star \text{list}(x, vs) \star |vs| > 10) \text{ LClient}(x) (\text{False})$$

This implicit loss of non-terminating branches can be characterised informally as follows:

(O5) if the post-conditions do not cover all paths allowed by the pre-condition,
then the “gap” is non-terminating.

In this case, the pre-condition implies that $|vs| \in \mathbb{N}$ and the post-conditions cover the cases where $|vs| \leq 10$, leaving the gap when $|vs| > 10$, for which we provably have client non-termination.

In general, there are cases when non-terminating branches cannot be captured by ESL specifications. For example, if the code branches on a value that does not originate from the pre-condition and if one of the resulting branches does not terminate, and if the code can also terminate successfully, then the non-terminating branch will be implicit in the pre-condition, but no gap in the sense

of (O5) will be present. This is illustrated by the code and specification below, where the pre- and the post-condition are the same, but a non-terminating path still exists.

```
( x = null ) x := nondet; if ( x > 42 ) { while ( true ) { skip } } else { x := null } ( x = null )
```

Exact Verification in Gillian. We have adapted the OX verification of Gillian to EX verification of recursive functions by, essentially, disallowing use of forward consequence and replacing it with equivalence. The details of this adaptation are intricately tied to the parametricity of Gillian and are, therefore, beyond the scope of this paper.

We have implemented, exactly specified, and verified a number of iterative and recursive list algorithms, including list-length, list-free, list-reverse, list-copy, list-append, and list membership. For each, we provided several specifications with different degrees of abstraction, using the various list predicates given in this section. To illustrate, the EX specifications of the recursive list-length algorithm given in the introduction and repeated in this section are written in Gillian as follows:

```
( ( x == #x ) * list(#x, #n) ) with variant: #n
  function LLen(x) { ... }
  ( list(#x, ret) )

( ( x == #x ) * list(#x, #xs, #vs) ) with variant: len #vs
  function LLen(x) { ... }
  ( list(#x, #xs, #vs) * (ret == len #vs) )
```

where the variables prefixed with the hash symbol denote logical variables and len denotes the list-length operator. When it comes to additional annotation, recursive function pre-conditions and loop invariants (which are always needed for semi-automatic verification) need to be equipped with an explicit variant, which corresponds to the decreasing measure (#n and len #vs in the above specifications), whereas predicate folding and unfolding is automatic.

6 COMPOSITIONAL SYMBOLIC EXECUTION WITH ESL SPECIFICATIONS

Our motivation for ESL came from Gillian, a multi-language symbolic analysis platform. On the way to bringing ESL results back to Gillian, we now turn to symbolic execution. Specifically, inspired by Gillian, we introduce a compositional symbolic execution semantics (CSE), which handles function calls by using UX (that is, ISL or ESL) specifications, for our demonstrator language and prove that that this CSE enjoys true bug-finding. We are not aware of a similar proof of function compositionality for symbolic execution in the literature. We conclude by highlighting the interplay between abstractions, which can hide information, and symbolic execution, which cannot.

Symbolic Values, Expressions and Assertions. We introduce the symbolic constructs on which our CSE and correctness results depend, starting from symbolic values, $\hat{v} \in \text{SVal}$, which are built from concrete values and symbolic variables, $\hat{x} \in \text{SVar}$:

$$\hat{v} \in \text{SVal} \triangleq v \mid \hat{x} \mid \hat{v} + \hat{v} \mid \hat{v} - \hat{v} \mid \dots \mid \hat{v} = \hat{v} \mid \neg \hat{v} \mid \hat{v} \wedge \hat{v} \mid \dots \mid \hat{v} \cdot \hat{v} \mid \hat{v} : \hat{v} \mid \dots$$

Symbolic expressions are defined analogously to logical expressions, with the only difference being in them having symbolic variables (via symbolic values), instead of logical variables:

$$\hat{e} \in \text{SExp} \triangleq \hat{v} \mid x \mid \hat{e} + \hat{e} \mid \hat{e} - \hat{e} \mid \dots \mid \hat{e} = \hat{e} \mid \neg \hat{e} \mid \hat{e} \wedge \hat{e} \mid \dots \mid \hat{e} \cdot \hat{e} \mid \hat{e} : \hat{e} \mid \dots$$

and symbolic assertions are defined analogously to logical assertions (cf. Def. 4.1).

Definition 6.1 (Symbolic Assertions). Symbolic assertions are defined as follows:

$$\hat{\pi} \in \text{SBAsrt} \triangleq \hat{e}_1 = \hat{e}_2 \mid \hat{e}_1 < \hat{e}_2 \mid \hat{e} \in X \mid \dots \mid \neg \hat{\pi} \mid \hat{\pi}_1 \Rightarrow \hat{\pi}_2$$

$$\hat{P} \in \text{SAst} \triangleq \hat{\pi} \mid \text{False} \mid \hat{P}_1 \Rightarrow \hat{P}_2 \mid \exists \hat{x}. \hat{P} \mid \text{emp} \mid \hat{e}_1 \mapsto \hat{e}_2 \mid \hat{e} \mapsto \emptyset \mid \hat{P}_1 \star \hat{P}_2 \mid \bigotimes_{\hat{e}_1 \leq \hat{x} < \hat{e}_2} \hat{P}$$

where $\hat{e}, \hat{e}_1, \hat{e}_2 \in \text{SExp}$, $X \subseteq \text{Val}$, and $\hat{x} \in \text{SVar}$.

Notation. For convenience, we introduce a generic function $\text{sv}(X)$, which collects the symbolic variables of a given construct X (e.g., a symbolic expression or assertion, and later store or memory), and write $X \subseteq \varepsilon$ to denote $\text{sv}(X) \subseteq \text{dom}(\varepsilon)$.

For symbolic assertion satisfiability, we introduce symbolic interpretations, $\varepsilon : \text{SVar} \rightarrow_{\text{fin}} \text{Val}$, which are partial finite mappings from symbolic variables to values, and lift them to symbolic values, overloading the ε notation. We also lift symbolic interpretations to a number of other symbolic constructs, always overloading the ε notation.

Definition 6.2 (Symbolic Assertion Satisfiability and Models). The symbolic satisfiability relation is first defined for Boolean symbolic assertions, denoted by $\varepsilon, s \models \hat{\pi}$, and is then lifted to arbitrary symbolic assertions, denoted by $\varepsilon, \sigma \models \hat{P}$, analogously to Def. 4.2. The set of models of a symbolic assertion \hat{P} is defined in the standard way: $\text{Mod}(\hat{P}) \stackrel{\text{def}}{=} \{\sigma \mid \exists \varepsilon. \varepsilon, \sigma \models \hat{P}\}$, and we use the notation $\sigma \models \hat{P}$ to denote $\sigma \in \text{Mod}(\hat{P})$.

Relating Symbolic and Logical Assertions. Observe that symbolic and logical assertions are isomorphic. We provide a mechanism for straightforwardly moving from logical to symbolic assertions by introducing symbolic substitutions, $\hat{\theta} : \text{LVar} \rightarrow_{\text{fin}} \text{SVal}$, which are partial finite mappings from logical variables to symbolic expressions, and lift them to logical values, expressions, and assertions in the standard way (the latter two maintain program variables), overloading the $\hat{\theta}$ notation, as for interpretations. We write $P\hat{\theta}$ to denote $\hat{\theta}(P)$ to keep in line with the common notation for substitutions. We extend interpretations to symbolic substitutions by interpreting their co-domain, yielding concrete substitutions: $\varepsilon : (\text{LVar} \rightarrow_{\text{fin}} \text{SVal}) \rightarrow_{\text{fin}} (\text{LVar} \rightarrow_{\text{fin}} \text{Val})$.

LEMMA 6.3. *The following properties can be proven by induction on E and P , respectively:*

$$\begin{aligned} \text{sv}(\hat{\theta}) \subseteq \varepsilon &\implies \llbracket E \rrbracket_{\varepsilon(\hat{\theta}), s} = \llbracket \hat{\theta}(E) \rrbracket_{\varepsilon, s} & (1) \\ \varepsilon(\hat{\theta}), \sigma \models P &\iff \varepsilon, \sigma \models P\hat{\theta} & (2) \end{aligned}$$

Symbolic States. Symbolic states, $\hat{\sigma} = (\hat{s}, \hat{h}, \hat{\pi})$, comprise: a symbolic store, $\hat{s} : \text{PVar} \rightarrow_{\text{fin}} \text{SVal}$, mapping program variables to symbolic values; a symbolic heap, $\hat{h} : \text{SVal} \rightarrow_{\text{fin}} (\text{SVal} \uplus \emptyset)$; and a path condition, $\hat{\pi} \in \text{SVal}$, capturing constraints imposed on symbolic variables during execution.

We extend interpretations to symbolic stores (by interpreting the co-domain) and to heaps (by interpreting both the domain and co-domain), requiring the following well-formedness constraints:

$$\begin{aligned} \mathcal{W}f_{\varepsilon}(\hat{s}) &\iff \hat{s} \subseteq \varepsilon \wedge \nexists \varepsilon(\text{codom}(\hat{s})) \\ \mathcal{W}f_{\varepsilon}(\hat{h}) &\iff \hat{h} \subseteq \varepsilon \wedge \varepsilon(\text{dom}(\hat{h})) \subset \mathbb{N} \wedge |\text{dom}(\hat{h})| = |\varepsilon(\text{dom}(\hat{h}))| \wedge \nexists \varepsilon(\text{codom}(\hat{h})) \end{aligned}$$

These constraints expectedly require non-faulting evaluation after interpretation (e.g., disallowing interpretations that assign incorrectly typed values), but also that the interpretation of the heap domain yields disjoint addresses. We extend well-formedness to symbolic states:

$$\mathcal{W}f((\hat{s}, \hat{h}, \hat{\pi})) \iff \hat{\pi} \text{ SAT} \wedge (\forall \varepsilon. \hat{s}, \hat{h} \subseteq \varepsilon \wedge \varepsilon(\hat{\pi}) = \text{true} \implies \mathcal{W}f_{\varepsilon}(\hat{s}) \wedge \mathcal{W}f_{\varepsilon}(\hat{h}))$$

requiring store and heap well-formedness for any interpretation that can interpret all of the components and validates $\hat{\pi}$, which also has to be satisfiable. Analogously, we define $\mathcal{W}f_{\hat{\pi}}(\hat{s})$ and $\mathcal{W}f_{\hat{\pi}}(\hat{h})$. We extend interpretations to well-formed symbolic states as follows:

$$\varepsilon((\hat{s}, \hat{h}, \hat{\pi})) \stackrel{\text{def}}{=} \begin{cases} (\varepsilon(\hat{s}), \varepsilon(\hat{h})), & \text{if } \mathcal{W}f_{\varepsilon}(\hat{s}) \wedge \mathcal{W}f_{\varepsilon}(\hat{h}) \wedge \varepsilon(\hat{\pi}) = \text{true} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

and define symbolic state models and satisfiability between symbolic and concrete states:

$$\text{Mod}(\hat{\sigma}) \stackrel{\text{def}}{=} \{\sigma \mid \exists \varepsilon. \varepsilon(\hat{\sigma}) = \sigma\} \quad \sigma \models \hat{\sigma} \iff \sigma \in \text{Mod}(\hat{\sigma})$$

$$\begin{array}{c}
\text{ASSIGN} \\
\frac{\llbracket E \rrbracket_s^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{s}' = \hat{s}[x \mapsto \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), x := E \Downarrow_{\Gamma} ok : (\hat{s}', \hat{h}, \hat{\pi}')} \\
\\
\text{FREE} \\
\frac{\llbracket E \rrbracket_s^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \hat{v}_m \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}) \wedge \hat{\pi}'}{\text{SAT}(\hat{\pi}'') \quad \hat{h}' = \hat{h}[\hat{v}_l \mapsto \emptyset]} \\
(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} ok : (\hat{s}, \hat{h}', \hat{\pi}'') \\
\\
\text{SEQ} \\
\frac{\hat{\sigma}, C_1 \Downarrow_{\Gamma} ok : \hat{\sigma}' \quad \hat{\sigma}', C_2 \Downarrow_{\Gamma} o : \hat{\sigma}''}{\hat{\sigma}, C_1; C_2 \Downarrow_{\Gamma} o : \hat{\sigma}''} \\
\\
\text{MUTATE-ERR-USE-AFTER-FREE} \\
\frac{\llbracket E_1 \rrbracket_s^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \emptyset \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \hat{v}_{err} = [\text{"UseAfterFree"}, \text{str}(E_1), \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} err : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{MUTATE-ERR-MISSING} \\
\frac{\llbracket E_1 \rrbracket_s^{\hat{\pi}} \Downarrow \hat{v}_1^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{v}_1 \in \mathbb{N} \wedge \hat{v}_1 \notin \text{dom}(\hat{h}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \hat{v}_{err} = [\text{"MissingCell"}, \text{str}(E_1), \hat{v}_1]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} miss : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{MUTATE} \\
\frac{\llbracket E_1 \rrbracket_s^{\hat{\pi}} \Downarrow \hat{v}_1^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \hat{v}_m \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}_1) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \llbracket E_2 \rrbracket_s^{\hat{\pi}''} \Downarrow \hat{v}_2^{\hat{\pi}'''} \quad \hat{h}' = \hat{h}[\hat{v}_l \mapsto \hat{v}_2]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} ok : (\hat{s}, \hat{h}', \hat{\pi}''')}
\end{array}$$

Fig. 9. Symbolic operational semantics, selection of rules, where $\hat{s}_{err} \triangleq \hat{s}[\text{err} \rightarrow \hat{v}_{err}]$

as well as three models-based relations, $\subseteq_{\mathcal{M}}$, $\supseteq_{\mathcal{M}}$, and $=_{\mathcal{M}}$, between symbolic states and assertions:

$$\begin{array}{ll}
\hat{\sigma} \subseteq_{\mathcal{M}} \hat{P} \iff \forall \varepsilon, \sigma. \sigma = \varepsilon(\hat{\sigma}) \Rightarrow \varepsilon, \sigma \models \hat{P} & (\text{implying } \text{Mod}(\hat{\sigma}) \subseteq \text{Mod}(\hat{P})) \\
\hat{\sigma} \supseteq_{\mathcal{M}} \hat{P} \iff \forall \varepsilon, \sigma. \sigma = \varepsilon(\hat{\sigma}) \Leftarrow \varepsilon, \sigma \models \hat{P} & (\text{implying } \text{Mod}(\hat{\sigma}) \supseteq \text{Mod}(\hat{P})) \\
\hat{\sigma} =_{\mathcal{M}} \hat{P} \iff \hat{\sigma} \subseteq_{\mathcal{M}} \hat{P} \wedge \hat{\sigma} \supseteq_{\mathcal{M}} \hat{P} & (\text{implying } \text{Mod}(\hat{\sigma}) = \text{Mod}(\hat{P}))
\end{array}$$

the second of which, $\supseteq_{\mathcal{M}}$ is essential for our correctness proof. In particular, it states that the state satisfying an assertion is uniquely determined by the interpretation. We discuss the ramifications of this requirement shortly.

Compositional Symbolic Semantics. For our CSE, we assume to have the symbolic expression evaluation relation, $\llbracket E \rrbracket_s^{\hat{\pi}} \Downarrow \hat{w}^{\hat{\pi}'}$, where \hat{w} denotes either a symbolic value or $\not\downarrow$, and the (satisfiable) output path condition, $\hat{\pi}' \Rightarrow \hat{\pi}$, may extend $\hat{\pi}$ with additional conditions under which the evaluation branches (e.g., division branching on denominator equalling zero). We keep symbolic expression evaluation opaque, as it carries little insight.

We provide a big-step symbolic semantics for the simple demonstrator programming language used for ESL, with two differences. First, we disallow dynamic memory allocation, as that would require a more complex representation of symbolic states; instead, we allow only allocation of concrete size. Second, we do not handle while loops, as that is orthogonal to our goals and would introduce clutter, and assume that they have been transformed into recursive functions.

The symbolic operational semantics uses judgements of the form $\hat{\sigma}, C \Downarrow_{\Gamma} o : \hat{\sigma}'$, meaning that, given specification context Γ and starting from state $\hat{\sigma}$, the execution of command C results in outcome o (*ok*, *err*, or *miss*) and state $\hat{\sigma}'$. We present a selection of rules in Figure 9 that illustrate the main points of the reasoning; the full semantics is given in Appendix G. The function call rule is introduced separately shortly due to its complexity.

We highlight three of the mutation rules, as they are representative of the single-trace reasoning that we use for the symbolic execution. In particular, [MUTATE] first evaluates E_1 , obtaining a symbolic value \hat{v}_1 , and then checks if it is possible for \hat{v}_1 to equal a non-freed address in the heap, \hat{v}_l , in which cases it takes that branch by adding the appropriate equality to the path condition, evaluates E_2 to obtain \hat{v}_2 , and updates the value of \hat{v}_l in the heap to \hat{v}_2 . Similarly, the [MUTATE-ERR-USE-AFTER-FREE] captures the branches in which \hat{v}_1 equals a freed address in the heap, whereas the [MUTATE-ERR-MISSING] rule covers the branch in which it is not in the heap at all. Note that

it is not necessary to add $\hat{v}_l \in \text{dom}(\hat{h})$ to the final path condition in non-missing rules, as that is guaranteed by the well-formedness of the initial symbolic state.

Treatment of Function Calls. Consider how function specifications are applied in ESL proof sketches, given the proof rules of §4.3 and the examples of §5. In particular, if we have the command $y := f(\vec{E})$, specification $(\vec{x} = \vec{x} * P) f(\vec{x}) (ok : Q_{ok})$ (assuming no faulting executions for simplicity), and current (logical) assertion A , this process can roughly be split into the following steps:

- (L1) using logical equivalence⁷ to massage A to match the function pre-condition plus some frame, that is, into the form $(\vec{E} = \vec{x} * P) * F$;
- (L2) framing F off;
- (L3) using the function call rule to replace the matched $\vec{E} = \vec{x} * P$ with $Q_{ok}[y/\text{ret}]$; and
- (L4) framing F back on.

On the other hand, our approach to performing the same function call in symbolic execution with current state $\hat{\sigma}$ is as follows:

- (S1) understand which part of $\hat{\sigma}$ corresponds to $\vec{E} = \vec{x} * P$ and consume it, leaving only the part that corresponds to the frame, $\hat{\sigma}_F$;
- (S2) extend $\hat{\sigma}_F$ with the symbolic state $\hat{\sigma}_Q$ corresponding to $Q_{ok}[y/\text{ret}]$, that is, produce $\hat{\sigma}_Q$ in $\hat{\sigma}_F$.

These two approaches intuitively correspond to each other; the main difference is that in logic the frame is removed, while in symbolic execution it is maintained. Onward, as is standard for symbolic execution tools, we restrict the set of allowed assertions in the specifications to exclude spatial negation, implication, and iterative separating conjunction. Following the approach of [33], we assume that pre- and post-conditions do not have explicit existential quantification, treating logical variables in the pre-condition as universally quantified and logical variables in the post-condition that are not in the pre-condition as implicitly existentially quantified. Any ESL specification can be trivially transformed into this format while preserving equivalence.

With these constraints in place, we give the function call rule for our CSE:

$$\begin{array}{l}
 \llbracket \vec{E} \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \vec{\sigma}^{\hat{\pi}'} \quad \text{evaluate function parameters} \\
 (\vec{x} = \vec{x} * P) f(\vec{x}) (ok : Q_{ok}) (err : Q_{err}) \in \Gamma \quad \text{get function specification} \\
 \hat{\theta} = [\vec{x} \mapsto \vec{\sigma}] \quad \text{create initial substitution} \\
 \text{matchAndConsume}(P, \hat{\theta}, (\hat{s}, \hat{h}, \hat{\pi}')) \rightsquigarrow (\hat{\theta}', \hat{h}_p, (\hat{s}, \hat{h}_f, \hat{\pi}''))^{ok} \quad \text{consume pre-condition (step (S1))} \\
 r, \hat{r} \text{ fresh} \quad \text{generate fresh vars for return value} \\
 Q'_{ok} = Q_{ok}[r/\text{ret}] \text{ and } \hat{\theta}'' = \hat{\theta}''[r \mapsto \hat{r}] \quad \text{set up return value} \\
 \text{produce}(Q'_{ok}, \hat{\theta}'', (\hat{s}, \hat{h}_f, \hat{\pi}'')) \rightsquigarrow (\hat{\theta}''', \hat{h}_q, (\hat{s}, \hat{h}', \hat{\pi}'''))^{ok} \quad \text{produce post-condition (step (S2))} \\
 \hline
 (\hat{s}, \hat{h}, \hat{\pi}), y := f(\vec{E}) \Downarrow_{\Gamma} ok : (\hat{s}[y \mapsto \hat{r}], \hat{h}', \hat{\pi}''')
 \end{array}$$

For convenience, we use ESL specifications in the presentation; in general, any UX specification can be used. This rule uses two auxiliary functions, `matchAndConsume` (in charge of (S1)) and `produce` (in charge of (S2)), which we present axiomatically.⁸ In particular, we require that both

$$\text{matchAndConsume}(P, \hat{\theta}, (\hat{s}, \hat{h}, \hat{\pi})) \rightsquigarrow (\hat{\theta}', \hat{h}_p, (\hat{s}', \hat{h}_f, \hat{\pi}'))^o$$

and

$$\text{produce}(P, \hat{\theta}, (\hat{s}, \hat{h}_f, \hat{\pi})) \rightsquigarrow (\hat{\theta}', \hat{h}_p, (\hat{s}', \hat{h}, \hat{\pi}'))^o$$

satisfy the following properties for successful execution (when $o = ok$):

⁷Or consequence/backward consequence in SL/ISL, this is the only function-call difference between the three logics.

⁸In Gillian, both functions are implemented parametrically on the memory model of the language under analysis, forming a parametric spatial entailment engine. We believe that their complexity deserves a separate publication.

- (P1) $\text{dom}(\hat{\theta}) \subseteq \text{fv}(P) \wedge \hat{\theta}' \geq \hat{\theta} \wedge \text{dom}(\hat{\theta}') = \text{fv}(P)$: this means that, given the initial bindings of $\hat{\theta}$, both functions extend $\hat{\theta}$ to cover all of the logical variables of P ; `matchAndConsume` learns them as part of the matching process, and `produce` generates fresh symbolic variables for the existentials of P , as per (P8) below;
- (P2) $\hat{s}' = \hat{s}$: this means that the store cannot be modified by the consumption or production of assertions, which is expected as we do not treat program variables as resource;
- (P3) $\hat{h} = \hat{h}_p \uplus \hat{h}_f$: for `matchAndConsume`, this means that it syntactically splits the initial heap, \hat{h} , into the heap that corresponds to the spatial part of P , \hat{h}_p , and the remaining frame, \hat{h}_f ; for `produce`, this means that it syntactically extends the frame, \hat{h}_f , with the heap corresponding to the spatial part of P , \hat{h}_p , resulting in the final heap, \hat{h} ;
- (P4) $\pi' \Rightarrow \pi$: this means that the path condition may only get strengthened, an expected property of symbolic execution: `matchAndConsume` may strengthen it to capture that specific branch of the matching; and `produce` may strengthen it with pure assertions that are part of P ;
- (P5) $\mathcal{W}f(\hat{s}, \hat{h}, \hat{\pi}')$: this captures that both initial and final states of both functions are well-formed, relying on (P4) and the monotonicity properties of $\mathcal{W}f$; as a consequence, this also means that π' is satisfiable;
- (P6) $(\emptyset, \hat{h}_p, \hat{\pi}') \supseteq_{\mathcal{M}} P \hat{\theta}' \star \hat{\pi}'$: this property links the consumed/produced assertion P to the corresponding symbolic state, requiring that the $\hat{\theta}$ covers P , in the style of UX backward consequence. This is a rare point where OX, UX, and EX symbolic execution differ: in particular, OX verification requires $\subseteq_{\mathcal{M}}$, UX true bug-finding requires $\supseteq_{\mathcal{M}}$, and EX reasoning requires $=_{\mathcal{M}}$. As a consequence, we obtain the expected satisfiability between P and the interpretation of its corresponding symbolic state, which holds for all three scenarios:
- (P7) $\forall \varepsilon. \varepsilon(\hat{\pi}') = \text{true} \wedge \hat{\theta}', \hat{s}, \hat{h} \subseteq \varepsilon \implies \varepsilon(\hat{\theta}'), \varepsilon(\hat{s}, \hat{h}_p, \hat{\pi}') \models P$.

Finally, for `produce`, we require in addition that the logical variables of P not in the domain of $\hat{\theta}$ (that is, the existentials of P) are afterwards mapped to fresh symbolic variables

- (P8) $\hat{\theta}' = \hat{\theta}[\vec{y} \mapsto \vec{\tilde{y}}]$, where $\{\vec{y}\} = \text{fv}(P) \setminus \text{dom}(\hat{\theta})$ and $\vec{\tilde{y}}$ fresh.

The symbolic execution also has function call rules that handle erroneous and missing executions. The erroneous ones that arise due to function parameter evaluation failing or the error post-condition being produced are standard and are therefore delegated to Appendix G. On the other hand, the case in which `matchAndConsume` fails carries additional insight. It means that it is not possible to apply the given specification in the given state, which could be due to, for example, the specification being incomplete, the symbolic state not having the required resource, or the code being incorrect. In all of those cases, there can be no guarantees regarding the behaviour of the corresponding concrete execution, and there are several approaches to handling this issue. In program logic, if a function specification cannot be applied, the proof cannot continue; we take this approach, which amounts to simply not having the rules for `matchAndConsume` failing in our CSE. Alternatively, we could instead attempt to symbolically execute the body of the function in such cases, which would be a sound solution. Finally, we note that there are no error/missing rules regarding `produce` because if `matchAndConsume` succeeds, then `produce` will also succeed due to the validity of the used specification.

True Bug-finding of Compositional Symbolic Execution. We prove that our CSE respects backward completeness, a property corresponding to UX validity in ESL, and therefore, by consequence, preserves true bug-finding.⁹ Here, we give a high-level overview of the proof; the full

⁹Note that terminology is used inconsistently throughout the literature. What we call completeness is called “correctness” by de Boer and Bonsangue [12], “soundness” by Godefroid et al. [24], and “completeness” by Baldoni et al. [5].

details can be found in Appendix G.2. To our knowledge, this is the first proof of its kind for a symbolic execution that can use UX/EX function specifications coming from a compositional program logic.¹⁰

THEOREM 6.4 (BACKWARD COMPLETENESS: SYMBOLIC EXECUTION).

$$\hat{\sigma}, C \Downarrow_{\Gamma} o : \hat{\sigma}' \wedge \models (\gamma, \Gamma) \wedge \sigma' \models \hat{\sigma}' \implies \exists \sigma. \sigma \models \hat{\sigma} \wedge \sigma, C \Downarrow_{\gamma} o : \sigma'$$

The correspondence between backward completeness and UX soundness for ESL is evident: $\hat{\sigma}, C \Downarrow_{\Gamma} o : \hat{\sigma}'$ corresponds to $\Gamma \vdash (P) \ C \ (Q)$ of Theorem 4.12, $\models (\gamma, \Gamma)$ to the same part of Definition 4.11, and the rest to UX validity of Definition 4.5, frame notwithstanding.

Abstractions for True Bug-finding. So far, the constraints placed on the allowed specifications are standard. However, property (P6) can be broken by existential quantification, disjunction, and abstractions that are not strictly exact (that is, that hide information). We focus the discussion on abstractions and illustrate the issue with an example; disjunction is handled analogously and existential quantification has been taken care of given the already placed constraints.

Consider the list-length specification from §5: $(x = x \star \text{list}(x, n)) \ \text{LLen}(x) \ (\text{list}(x, n) \star \text{ret} = n)$, where the $\text{list}(x, n)$ predicate hides information about the list node addresses and values, and consider the symbolic state $\hat{\sigma} = (\hat{s}, \hat{h}, \hat{\pi})$, where $\hat{s} \equiv \{x \mapsto \hat{x}, y \mapsto \text{null}\}$, $\hat{h} \equiv \{\hat{x} \mapsto 42, \hat{x} + 1 \mapsto \text{null}\}$, and $\hat{\pi} \equiv \hat{x} \in \mathbb{N}$, in which there is a linked list at \hat{x} consisting of one node carrying the value 42. Further consider the command $y := \text{LLen}(x)$. Given the structure of $\hat{\sigma}$, one would expect to be able to apply the given specification, but this is not possible in a way that preserves true bug-finding. In particular, the (P6) requirement for matchAndConsume amounts to

$$(\emptyset, \{\hat{x} \mapsto 42, \hat{x} + 1 \mapsto \text{null}\}, \hat{x} \in \mathbb{N}) \supseteq_{\mathcal{M}} \text{list}(\hat{x}, 1) \star \hat{x} \in \mathbb{N}$$

but this does not hold, as, for example, the concrete state $(\emptyset, \{0 \mapsto 43, 1 \mapsto \text{null}\})$ is in the models of the right-hand side, but not in the models of the left-hand side, with the discrepancy being in the node values (42 vs. 43), which is precisely the information hidden by the list predicate.

This means that compositional symbolic execution for true-bug finding may only use specifications that contain strictly exact abstractions, such as $\text{list}(x, xs, vs)$. In particular, for that predicate and the relevant list-length specification

$$(x = x \star \text{list}(x, xs, vs)) \ \text{LLen}(x) \ (\text{list}(x, xs, vs) \star \text{ret} = |xs|)$$

the (P6) requirement amounts to:

$$(\emptyset, \{\hat{x} \mapsto 42, \hat{x} + 1 \mapsto \text{null}\}, \hat{x} \in \mathbb{N}) \supseteq_{\mathcal{M}} \text{list}(\hat{x}, [\hat{x}], [42]) \star \hat{x} \in \mathbb{N}$$

and the above-mentioned issue no longer exists. We believe this to be the boundary of sound use of abstractions for true-bug finding with standard symbolic states.

One way of crossing this boundary would be to have symbolic states of the form $(\hat{\sigma}, \hat{h}, \hat{\Delta}, \hat{\pi})$, where the new component, $\hat{\Delta}$, contains a list of predicates, and also extend the symbolic execution with commands for unfolding and folding predicates. This, in particular, allows information hiding in symbolic states and is both what happens in the implementation of Gillian OX and EX verification and what is required to potentially model dynamic memory allocation. In this context, the (P6) requirement would hold for a broader class of symbolic states and assertions. For example, we would have that:

$$(\emptyset, \emptyset, [\text{list}(\hat{x}, 1)], \hat{x} \in \mathbb{N}) \supseteq_{\mathcal{M}} \text{list}(\hat{x}, 1) \star \hat{x} \in \mathbb{N}$$

letting us believe that there is room for soundly bringing in more abstraction to symbolic execution while maintaining true bug-finding. We leave a deeper exploration of this extension for future work.

¹⁰The literature contains examples of symbolic execution with function summaries (e.g. [1, 23, 25, 27, 32, 47]), but those either come without a soundness proof or use first-order summaries that do not talk about the heap.

7 CONCLUSIONS AND FURTHER WORK

We have introduced exact reasoning for analysing heap-manipulating programs by presenting an exact separation logic, ESL. Exact specifications provide a bridge between verification and true bug-finding, as they can be soundly used for both: they guarantee the absence of bugs for success post-conditions and, at the same time, all bugs exposed in error post-conditions are true. ESL supports reasoning about mutually recursive functions and comes with a frame-preserving soundness result that transfers straightforwardly to UX and OX separation logics, thus demonstrating, for the first time, functional compositionality for UX reasoning.

We have verified exact specifications for a number of illustrative examples, showing how ESL can be used to reason about data-structure libraries, language errors, mutual recursion and non-termination. In particular, we verify exact specifications for list algorithms using familiar inductive predicates for singly-linked lists, demonstrating that abstraction can be soundly used in exact and UX reasoning. We emphasise the distinction between hiding information through existential quantification, which can be used with exact and UX reasoning, and losing information through forwards consequence that can only be used in OX reasoning. We have adapted the OX verification of Gillian [33] to exact verification, and have verified the examples presented here. As future work, we will explore the applicability of Gillian's exact verification to real-world code: in particular, we believe that the parts of the AWS codebase that have already been OX-verified by Gillian [33] can be adapted to exact verification.

To demonstrate overall viability of exact verification for true bug-finding, we have introduced a compositional symbolic execution semantics that is able to call functions described using exact specifications, precisely pinpointing when such specifications are applicable (property (P6) of `matchAndConsume` and produce introduced §6): for example, the list algorithms require that the specifications are defined using strictly exact list-predicate assertions. Our ultimate aim is to unify OX, UX and exact reasoning in Gillian, underpinned by the appropriately parametric and monadic version of compositional symbolic execution presented here.

REFERENCES

- [1] S. Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), 367–381. https://doi.org/10.1007/978-3-540-78800-3_28
- [2] Saswat Anand, Corina S. Pasareanu, and Willem Visser. 2009. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer* 11, 1 (2009), 53–67. <https://doi.org/10.1007/s10009-008-0090-1>
- [3] Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods Symposium (NFM)*, Vol. 7226. https://doi.org/10.1007/978-3-642-28891-3_2
- [4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019). <https://doi.org/10.1145/3360573>
- [5] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018). <https://doi.org/10.1145/3182657>
- [6] Stephen Brookes and Peter W. O’Hearn. 2016. Concurrent Separation Logic. *ACM SIGLOG News* 3, 3 (2016). <https://doi.org/10.1145/2984450.2984457>
- [7] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *Symposium on Logic in Computer Science (LICS)*. <https://doi.org/10.1109/LICS52264.2021.9470608>
- [8] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods Symposium (NFM)*. https://doi.org/10.1007/978-3-642-20398-5_33
- [9] Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *Principles of Programming Languages (POPL)*. <https://doi.org/10.1145/1480881.1480917>
- [10] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM (JACM)* 58, 6 (2011). <https://doi.org/10.1145/2049697.2049700>

- [11] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-662-44202-9_9
- [12] Frank S. de Boer and Marcello Bonsangue. 2021. Symbolic execution formally explained. *Formal Aspects of Computing* 33, 4-5 (2021). <https://doi.org/10.1007/s00165-020-00527-y>
- [13] Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods (SEFM)*. https://doi.org/10.1007/978-3-642-24690-6_12
- [14] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: Compositional reasoning for concurrent programs. In *Principles of Programming Languages (POPL)*. <https://doi.org/10.1145/2429069.2429104>
- [15] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-14107-2_24
- [16] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-language Platform for Symbolic Execution. In *Programming Language Design and Implementation (PLDI)*. <https://doi.org/10.1145/3385412.3386014>
- [17] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Principles and Practice of Declarative Programming (PPDP)*. <https://doi.org/10.1145/3236950.3236956>
- [18] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *PACMPL* 2, POPL (2018). <https://doi.org/10.1145/3158138>
- [19] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *PACMPL* 3, POPL (2019). <https://doi.org/10.1145/3290379>
- [20] José Fragoso Santos, Petar Maksimović, Sacha Élie Ayoun, and Philippa Gardner. 2020. Gillian: Compositional Symbolic Execution for All. *arXiv:2001.05059*
- [21] Philippa Gardner, Sergio Maffei, and Gareth David Smith. 2012. Towards a program logic for JavaScript. In *Principles of Programming Languages (POPL)*. <https://doi.org/10.1145/2103656.2103663>
- [22] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse, and Uri Zarfaty. 2008. Local Hoare reasoning about DOM. In *Principles of Database Systems (PODS)*. <https://doi.org/10.1145/1376916.1376953>
- [23] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Principles of Programming Languages (POPL)*. <https://doi.org/10.1145/1190216.1190226>
- [24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Conference on Programming Language Design and Implementation (PLDI)*. <https://doi.org/10.1145/1065010.1065036>
- [25] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *Principles of Programming Languages (POPL)*. <https://doi.org/10.1145/1706299.1706307>
- [26] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A true positives theorem for a static race detector. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). <https://doi.org/10.1145/3290370>
- [27] Benjamin Hillery, Eric Mercer, Neha Rungta, and Suzette Person. 2016. Exact Heap Summaries for Symbolic Execution. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. https://doi.org/10.1007/978-3-662-49122-5_10
- [28] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)* 12, 10 (1969). <https://doi.org/10.1145/363235.363259>
- [29] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods Symposium (NFM)*. https://doi.org/10.1007/978-3-642-20398-5_4
- [30] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Communications of the ACM (CACM)* 64, 4 (2021). <https://doi.org/10.1145/3418295>
- [31] Robbert Krebbers. 2016. A Formal C Memory Model for Separation Logic. *Journal of Automated Reasoning* 57, 4 (2016). <https://doi.org/10.1007/s10817-016-9369-1>
- [32] Yude Lin, Tim Miller, and Harald Sondergaard. 2015. Compositional Symbolic Execution Using Fine-Grained Summaries. In *Australasian Software Engineering Conference*. <https://doi.org/10.1109/ASWEC.2015.32>
- [33] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-030-81688-9_38
- [34] Petar Maksimović, José Fragoso Santos, Sacha Élie Ayoun, and Philippa Gardner. 2021. Gillian: A Multi-Language Platform for Unified Symbolic Analysis. <http://arxiv.org/abs/2105.14769>
- [35] Toby Murray, Pengbo Yan, and Gidon Ernst. 2021. Incremental Vulnerability Detection with Insecurity Separation Logic. *arXiv:2107.05225 [cs.PL]*

- [36] Gian Ntzik and Philippa Gardner. 2015. Reasoning about the POSIX file system: local update and global pathnames. In International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). <https://doi.org/10.1145/2814270.2814306>
- [37] Peter W. O’Hearn. 2019. Incorrectness Logic. Proceedings of the ACM on Programming Languages 4, POPL (2019). <https://doi.org/10.1145/3371078>
- [38] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In Computer Science Logic (CSL). https://doi.org/10.1007/3-540-44802-0_1
- [39] Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation Logic and Abstraction. In Principles of Programming Languages (POPL). <https://doi.org/10.1145/1040305.1040326>
- [40] 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 (CAV). https://doi.org/10.1007/978-3-030-53291-8_14
- [41] Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O’Hearn. 2022. Concurrent Incorrectness Separation Logic. Proceedings of the ACM on Programming Languages 6, POPL (2022). <https://doi.org/10.1145/3498695>
- [42] Azalea Raad, José Fragoso Santos, and Philippa Gardner. 2016. DOM: Specification and Client Reasoning. In Asian Symposium on Programming Languages and Systems (APLAS). https://doi.org/10.1007/978-3-319-47958-3_21
- [43] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In Logic in Computer Science (LICS). <https://doi.org/10.1109/LICS.2002.1029817>
- [44] Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. 2019. A Program Logic for First-Order Encapsulated WebAssembly. In European Conference on Object-Oriented Programming (ECOOP). <https://doi.org/10.4230/LIPICs.ECOOP.2019.9>
- [45] Glynn Winskel. 1993. The Formal Semantics of Programming Languages: An Introduction, Chapter 10. MIT Press, Cambridge, MA, USA.
- [46] Hongseok Yang. 2001. Local Reasoning for Stateful Programs. Ph.D. Dissertation. University of Illinois Urbana-Champaign.
- [47] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating precise and concise procedure summaries. In Principles of Programming Languages (POPL). <https://doi.org/10.1145/1328438.1328467>

$\sigma, \text{skip} \Downarrow_Y \sigma$	$\frac{\llbracket E \rrbracket_s = v}{(s, h), x := E \Downarrow_Y (s[x \rightarrow v], h)}$	$\frac{\llbracket E \rrbracket_s = \zeta \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), x := E \Downarrow_Y \text{err} : (s_{err}, h)}$
$\frac{n \in \mathbb{N}}{(s, h), x := \text{nondet} \Downarrow_Y (s[x \rightarrow n], h)}$	$\frac{\llbracket E \rrbracket_s = v \quad v_{err} = [\text{"Error"}, v]}{(s, h), \text{error}(E) \Downarrow_Y \text{err} : (s_{err}, h)}$	
$\frac{\llbracket E \rrbracket_s = \zeta \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), \text{error}(E) \Downarrow_Y \text{err} : (s_{err}, h)}$	$\frac{\llbracket E \rrbracket_\sigma = \text{true} \quad \sigma, C_1 \Downarrow_Y o : \sigma'}{\sigma, \text{if } (E) C_1 \text{ else } C_2 \Downarrow_Y o : \sigma'}$	$\frac{\llbracket E \rrbracket_\sigma = \text{true} \quad \sigma, C \Downarrow_Y \sigma'' \quad \sigma'', \text{while } (E) C \Downarrow_Y o : \sigma'}{\sigma, \text{while } (E) C \Downarrow_Y o : \sigma'}$
$\frac{\llbracket E \rrbracket_\sigma = \text{false} \quad \sigma, C_2 \Downarrow_Y o : \sigma'}{\sigma, \text{if } (E) C_1 \text{ else } C_2 \Downarrow_Y o : \sigma'}$	$\frac{\llbracket E \rrbracket_s = \zeta \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_Y \text{err} : (s_{err}, h)}$	
$\frac{v_{err} = [\text{"Type"}, \text{str}(E), v, \text{"Bool"}]}{(s, h), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_Y \text{err} : (s_{err}, h)}$	$\frac{\llbracket E \rrbracket_\sigma = \text{false}}{\sigma, \text{while } (E) C \Downarrow_Y \sigma}$	$\frac{\llbracket E \rrbracket_\sigma = \text{true} \quad \sigma, C \Downarrow_Y \sigma'' \quad \sigma'', \text{while } (E) C \Downarrow_Y o : \sigma'}{\sigma, \text{while } (E) C \Downarrow_Y o : \sigma'}$
$\frac{\llbracket E \rrbracket_s = \zeta \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), \text{while } (E) C \Downarrow_Y \text{err} : (s_{err}, h)}$	$\frac{\llbracket E \rrbracket_s = v \notin \mathbb{N} \quad v_{err} = [\text{"Type"}, \text{str}(E), v, \text{"Bool"}]}{(s, h), \text{while } (E) C \Downarrow_Y \text{err} : (s_{err}, h)}$	
$\frac{\llbracket E \rrbracket_s = \text{true} \quad \sigma, C \Downarrow_Y o : \sigma' \quad o \neq \text{ok}}{\sigma, \text{while } (E) C \Downarrow_Y o : \sigma'}$	$\frac{\sigma, C_1 \Downarrow_Y \sigma'' \quad \sigma'', C_2 \Downarrow_Y o : \sigma'}{\sigma, C_1; C_2 \Downarrow_Y \sigma'}$	$\frac{\sigma, C_1 \Downarrow_Y o : \sigma' \quad o \neq \text{ok}}{\sigma, C_1; C_2 \Downarrow_Y o : \sigma'}$
$\frac{f(\vec{x}) \{C; \text{return } E'\} \in \gamma \quad \begin{array}{l} \llbracket \vec{E} \rrbracket_s = \vec{v} \quad \text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\} \\ s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \\ (s_p, h), C \Downarrow_Y (s_q, h') \quad \llbracket E' \rrbracket_{s_q} = v' \end{array}}{(s, h), \gamma := f(\vec{E}) \Downarrow_Y (s[\gamma \rightarrow v'], h')}$	$\frac{f(\vec{x}) \{C; \text{return } E'\} \in \gamma \quad \begin{array}{l} \llbracket \vec{E} \rrbracket_s = \vec{v} \quad \text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\} \\ s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \\ (s_p, h), C \Downarrow_Y (s_q, h') \quad \llbracket E' \rrbracket_{s_q} = \zeta \\ v_{err} = [\text{"ExprEval"}, \text{str}(E')] \end{array}}{(s, h), \gamma := f(\vec{E}) \Downarrow_Y \text{err} : (s_{err}, h')}$	
$\frac{f(\vec{x}) \{C; \text{return } E'\} \in \gamma \quad \begin{array}{l} \llbracket \vec{E} \rrbracket_s = \vec{v} \quad \text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\} \\ s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \\ (s_p, h), C \Downarrow_Y o : (s_q, h') \quad o \neq \text{ok} \end{array}}{(s, h), \gamma := f(\vec{E}) \Downarrow_Y o : (s[\text{err} \rightarrow s_q(\text{err})], h')}$	$\frac{f(\vec{x}) \{C; \text{return } E'\} \in \gamma \quad \begin{array}{l} k \in \{1, \dots, n\} \quad (\llbracket E_i \rrbracket_s = v_i)_{i=1}^{k-1} \quad \llbracket E_k \rrbracket_s = \zeta \\ v_{err} = [\text{"ExprEval"}, \text{str}(E_k)] \end{array}}{(s, h), \gamma := f(E_1, \dots, E_n) \Downarrow_Y \text{err} : (s_{err}, h)}$	
$\frac{f \notin \text{dom}(\gamma) \quad v_{err} = [\text{"NoFunc"}, f]}{(s, h), x := f(\vec{E}) \Downarrow_Y \text{err} : (s_{err}, h)}$	$\frac{\llbracket E \rrbracket_s = n \quad h(n) = v}{(s, h), x := [E] \Downarrow_Y (s[x \rightarrow v], h)}$	$\frac{\llbracket E \rrbracket_s = \zeta \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), x := [E] \Downarrow_Y \text{err} : (s_{err}, h)}$
$\frac{\llbracket E \rrbracket_s = v \notin \mathbb{N} \quad v_{err} = [\text{"Type"}, \text{str}(E), v, \text{"Nat"}]}{(s, h), x := [E] \Downarrow_Y \text{err} : (s_{err}, h)}$	$\frac{\llbracket E \rrbracket_s = n \notin \text{dom}(h) \quad v_{err} = [\text{"MissingCell"}, \text{str}(E), n]}{(s, h), x := [E] \Downarrow_Y \text{miss} : (s_{err}, h)}$	
$\frac{\llbracket E \rrbracket_s = n \quad h(n) = \emptyset \quad v_{err} = [\text{"UseAfterFree"}, \text{str}(E), n]}{(s, h), x := [E] \Downarrow_Y \text{err} : (s_{err}, h)}$	$\frac{\llbracket E_1 \rrbracket_s = n \quad h(n) \in \text{Val} \quad \llbracket E_2 \rrbracket_s = v}{(s, h), [E_1] := E_2 \Downarrow_Y (s, h[n \mapsto v])}$	

$$\begin{array}{c}
\frac{\llbracket E_1 \rrbracket_s = \downarrow \quad v_{err} = [\text{"ExprEval"}, \text{str}(E_1)]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E_1 \rrbracket_s = n \notin \text{dom}(h) \quad v_{err} = [\text{"MissingCell"}, \text{str}(E_1), n]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{miss} : (s_{err}, h)} \\
\\
\frac{\llbracket E_1 \rrbracket_s = n \quad h(n) \in \text{Val} \quad \llbracket E_2 \rrbracket_s = \downarrow \quad v_{err} = [\text{"ExprEval"}, \text{str}(E_2)]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E \rrbracket_s = \downarrow \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), x := \text{new}(E) \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E_1 \rrbracket_s = v \notin \mathbb{N} \quad v_{err} = [\text{"Type"}, \text{str}(E_1), v, \text{"Nat"}]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E_1 \rrbracket_s = n \quad h(n) = \emptyset \quad v_{err} = [\text{"UseAfterFree"}, \text{str}(E_1), n]}{(s, h), [E_1] := E_2 \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E \rrbracket_s = n \quad (n' + i \notin \text{dom}(h))|_{0 \leq i < n} \quad h' = h[n' \mapsto \text{null}] \cdots [n' + n - 1 \mapsto \text{null}]}{(s, h), x := \text{new}(E) \Downarrow_Y (s[x \rightarrow n'], h')} \\
\\
\frac{\llbracket E \rrbracket_s = \downarrow \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), \text{free}(E) \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E \rrbracket_s = v \notin \mathbb{N} \quad v_{err} = [\text{"Type"}, \text{str}(E), v, \text{"Nat"}]}{(s, h), \text{free}(E) \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E \rrbracket_s = n \quad h(n) \in \text{Val} \quad v_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), \text{free}(E) \Downarrow_Y \text{err} : (s_{err}, h)} \\
\\
\frac{\llbracket E \rrbracket_s = n \notin \text{dom}(h) \quad v_{err} = [\text{"MissingCell"}, \text{str}(E), n]}{(s, h), \text{free}(E) \Downarrow_Y \text{miss} : (s_{err}, h)} \\
\\
\frac{\llbracket E \rrbracket_s = n \quad h(n) = \emptyset \quad v_{err} = [\text{"UseAfterFree"}, \text{str}(E), n]}{(s, h), \text{free}(E) \Downarrow_Y \text{err} : (s_{err}, h)}
\end{array}$$

where $s_{err} \triangleq s[\text{err} \rightarrow v_{err}]$.

B EXACT SEPARATION LOGIC

Definition B.1. The satisfiability relation, denoted $\theta, s \models \pi$ for pure assertions and $\theta, \sigma \models P$ for assertions, is defined as follows:

$\theta, s \models$	$\theta, (s, h) \models$	
$E_1 = E_2$	$\Leftrightarrow \llbracket E_1 = E_2 \rrbracket_{\theta, s} = \text{true}$	π
$E_1 < E_2$	$\Leftrightarrow \llbracket E_1 < E_2 \rrbracket_{\theta, s} = \text{true}$	False
$E \in X$	$\Leftrightarrow \llbracket E \rrbracket_{\theta, s} \in X$	$P_1 \Rightarrow P_2$
$\pi_1 \Rightarrow \pi_2$	$\Leftrightarrow \theta, s \models \pi_1 \Rightarrow \theta, s \models \pi_2$	$\exists x. P$
$\neg(E_1 = E_2)$	$\Leftrightarrow \llbracket E_1 = E_2 \rrbracket_{\theta, s} = \text{false}$	emp
$\neg(E_1 < E_2)$	$\Leftrightarrow \llbracket E_1 < E_2 \rrbracket_{\theta, s} = \text{false}$	$E_1 \mapsto E_2$
$\neg(E \in X)$	$\Leftrightarrow \llbracket E \rrbracket_{\theta, s} \notin X$	$E_1 \mapsto \emptyset$
$\neg(\pi_1 \Rightarrow \pi_2)$	$\Leftrightarrow \theta, s \models \pi_1 \wedge \theta, s \models \neg\pi_2$	$P_1 \star P_2$
$\neg\neg\pi$	$\Leftrightarrow \theta, s \models \pi$	$\otimes_{E_1 \leq x < E_2} P$
		$\Leftrightarrow \theta, s \models \pi \wedge h = \emptyset$ $\Leftrightarrow \text{never}$ $\Leftrightarrow \theta, (s, h) \models P_1 \Rightarrow \theta, (s, h) \models P_2$ $\Leftrightarrow \exists v \in \text{Val}. \theta[x \mapsto v], (s, h) \models P$ $\Leftrightarrow h = \emptyset$ $\Leftrightarrow h = \{\llbracket E_1 \rrbracket_{\theta, s} \mapsto \llbracket E_2 \rrbracket_{\theta, s}\}$ $\Leftrightarrow h = \{\llbracket E_1 \rrbracket_{\theta, s} \mapsto \emptyset\}$ $\Leftrightarrow \exists h_1, h_2. h = h_1 \uplus h_2 \wedge$ $\theta, (s, h_1) \models P_1 \wedge \theta, (s, h_2) \models P_2$ $\Leftrightarrow (i < k \wedge \exists h_i, \dots, h_{k-1}. h = \uplus_{j=i}^{k-1} h_j \wedge$ $\forall j. i \leq j < k \Rightarrow \theta, (s, h_j) \models P[j/x]) \vee$ $(i \geq k \wedge h = \emptyset), \text{ where } i = \llbracket E_1 \rrbracket_{\theta, s}, k = \llbracket E_2 \rrbracket_{\theta, s}$ and x is not featured in either E_1 or E_2 .

The complete rules of ESL are as follows (with $Q_{err} = \text{pre} \star \text{err} = E_{err}$):

SKIP $\Gamma \vdash (\text{emp}) \text{ skip } (\text{emp})$		NONDET $\frac{x \notin \text{pv}(E')}{\Gamma \vdash (x = E') \ x := \text{nondet } (E' \in \text{Val} \star x \in \mathbb{N})}$	
ASSIGN $\frac{x \notin \text{pv}(E') \quad \theta \triangleq [E'/x]}{\Gamma \vdash (x = E' \star E \in \text{Val}) \ x := E \ (E' \in \text{Val} \star x = E\theta)}$		ASSIGN-ERR $\frac{E_{err} \triangleq ["\text{ExprEval}", \text{str}(E)]}{\Gamma \vdash (x = E' \star E \notin \text{Val}) \ x := E \ (err : Q_{err})}$	
LOOKUP $\frac{x \notin \text{pv}(E') \quad \theta \triangleq [E'/x]}{\Gamma \vdash (x = E' \star E \mapsto E_1) \ x := [E] \ (E' \in \text{Val} \star x = E_1\theta \star E\theta \mapsto E_1\theta)}$			
LOOKUP-ERR-VAL $\frac{E_{err} \triangleq ["\text{ExprEval}", \text{str}(E)]}{\Gamma \vdash (x = E' \star E \notin \text{Val}) \ x := [E] \ (err : Q_{err})}$			
LOOKUP-ERR-TYPE $\frac{E_{err} \triangleq ["\text{Type}", \text{str}(E), E, "\text{Nat}"]}{\Gamma \vdash (x = E' \star E \in \text{Val} \star E \notin \mathbb{N}) \ x := [E] \ (err : Q_{err})}$		LOOKUP-ERR-USE-AFTER-FREE $\frac{E_{err} \triangleq ["\text{UseAfterFree}", \text{str}(E), E]}{\Gamma \vdash (x = E' \star E \mapsto \emptyset) \ x := [E] \ (err : Q_{err})}$	
MUTATE $\Gamma \vdash (E_1 \mapsto E \star E_2 \in \text{Val}) \ [E_1] := E_2 \ (E_1 \mapsto E_2 \star E \in \text{Val})$		MUTATE-ERR-VAL-1 $\frac{E_{err} \triangleq ["\text{ExprEval}", \text{str}(E_1)]}{\Gamma \vdash (E_1 \notin \text{Val}) \ [E_1] := E_2 \ (err : Q_{err})}$	
MUTATE-ERR-TYPE $\frac{E_{err} \triangleq ["\text{Type}", \text{str}(E_1), E_1, "\text{Nat}"]}{\Gamma \vdash (E_1 \in \text{Val} \star E_1 \notin \mathbb{N}) \ [E_1] := E_2 \ (err : Q_{err})}$		MUTATE-ERR-USE-AFTER-FREE $\frac{E_{err} \triangleq ["\text{UseAfterFree}", \text{str}(E_1), E_1]}{\Gamma \vdash (E_1 \mapsto \emptyset) \ [E_1] := E_2 \ (err : Q_{err})}$	

$$\begin{array}{c}
\text{MUTATE-ERR-VAL-2} \\
\frac{E_{err} \triangleq [\text{"ExprEval"}, \text{str}(E_2)]}{\Gamma \vdash (E_1 \mapsto E \star E_2 \not\in \text{Val}) [E_1] := E_2 \text{ (err : } Q_{err}\text{)}} \\
\\
\text{NEW} \\
\frac{x \notin \text{pv}(E') \quad \theta \triangleq [E'/x]}{\Gamma \vdash (x = E' \star E \in \mathbb{N}) \ x := \text{new}(E) \text{ (ok : } E' \in \text{Val} \star \otimes_{0 \leq i < \text{E}\theta} ((x+i) \mapsto \text{null}))} \\
\\
\text{NEW-ERR-EVAL} \qquad \text{NEW-ERR-TYPE} \\
\frac{E_{err} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (x = E' \star E \not\in \text{Val}) \ x := \text{new}(E) \text{ (err : } Q_{err}\text{)}} \quad \frac{E_{err} \triangleq [\text{"Type"}, \text{str}(E), E, \text{"Nat"}]}{\Gamma \vdash (x = E' \star E \in \text{Val} \star E \not\in \mathbb{N}) \ x := \text{new}(E) \text{ (err : } Q_{err}\text{)}} \\
\\
\text{FREE} \qquad \text{FREE-ERR-EVAL} \\
\frac{}{\Gamma \vdash (E \mapsto E') \text{ free}(E) \text{ (ok : } E' \in \text{Val} \star E \mapsto \emptyset\text{)}} \quad \frac{E_{err} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (E \not\in \text{Val}) \text{ free}(E) \text{ (err : } Q_{err}\text{)}} \\
\\
\text{FREE-ERR-TYPE} \qquad \text{FREE-ERR-USE-AFTER-FREE} \\
\frac{E_{err} \triangleq [\text{"Type"}, \text{str}(E), E, \text{"Nat"}]}{\Gamma \vdash (E \in \text{Val} \star E \not\in \mathbb{N}) \text{ free}(E) \text{ (err : } Q_{err}\text{)}} \quad \frac{E_{err} \triangleq [\text{"UseAfterFree"}, \text{str}(E), E]}{\Gamma \vdash (E \mapsto \emptyset) \text{ free}(E) \text{ (err : } Q_{err}\text{)}} \\
\\
\text{ERROR} \qquad \text{ERROR-ERR} \\
\frac{E_{err} \triangleq [\text{"Error"}, E]}{\Gamma \vdash (E \in \text{Val}) \text{ error}(E) \text{ (err : err = } E_{err}\text{)}} \quad \frac{E_{err} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (E \not\in \text{Val}) \text{ error}(E) \text{ (err : } Q_{err}\text{)}} \\
\\
\text{IF-THEN} \qquad \text{IF-ELSE} \\
\frac{\Gamma \vdash (P \wedge E) \ C_1 \ (Q)}{\Gamma \vdash (P \wedge E) \text{ if } (E) \ C_1 \text{ else } C_2 \ (Q)} \quad \frac{\Gamma \vdash (P \wedge \neg E) \ C_2 \ (Q)}{\Gamma \vdash (P \wedge \neg E) \text{ if } (E) \ C_1 \text{ else } C_2 \ (Q)} \\
\\
\text{IF-ERR-VAL} \\
\frac{E_{err} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (P \star E \not\in \text{Val}) \text{ if } (E) \ C_1 \text{ else } C_2 \text{ (err : } Q_{err}\text{)}} \\
\\
\text{IF-ERR-TYPE} \qquad \text{SEQ} \\
\frac{E_{err} \triangleq [\text{"Type"}, \text{str}(E), E, \text{"Bool"}]}{\Gamma \vdash (P \star E \in \text{Val} \star E \not\in \mathbb{B}) \text{ if } (E) \ C_1 \text{ else } C_2 \text{ (err : } Q_{err}\text{)}} \quad \frac{\Gamma \vdash (P) \ C_1 \text{ (ok : } R) \text{ (err : } Q_{err}^1\text{)} \quad \Gamma \vdash (R) \ C_2 \text{ (ok : } Q_{ok}\text{) (err : } Q_{err}^2\text{)}}{\Gamma \vdash (P) \ C_1; C_2 \text{ (ok : } Q_{ok}\text{) (err : } Q_{err}^1 \vee Q_{err}^2\text{)}} \\
\\
\text{WHILE-ITERATE} \\
\frac{\forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B} \quad P_\infty \triangleq \text{False} \quad \forall i \in \mathbb{N}. \Gamma \vdash (P_i \wedge E) \ C \text{ (ok : } P_{i+1}\text{) (err : } Q_i) \quad m \triangleq \min(\{i \in \mathbb{N} \cup \{\infty\} \mid \models P_i \Rightarrow \neg E\})}{\Gamma \vdash (P_0) \text{ while } (E) \ C \text{ (ok : } P_m\text{) (err : } \exists n < m. Q_n\text{)}} \\
\\
\text{WHILE-ITERATE-ERR-VAL} \\
\frac{\exists m \in \mathbb{N}. (\forall i \in \mathbb{N}^{<m}. \models P_i \Rightarrow E) \wedge \models P_m \Rightarrow E \notin \text{Val} \quad \forall i \in \mathbb{N}^{<m}. \Gamma \vdash (P_i) \ C \text{ (ok : } P_{i+1}\text{) (err : } Q_i) \quad E_{err} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (P_0) \text{ while } (E) \ C \text{ (err : } (\exists n < m. Q_n) \vee (P_m \star \text{err} = E_{err}))} \\
\\
\text{WHILE-ITERATE-ERR-TYPE} \\
\frac{\exists m \in \mathbb{N}. (\forall i \in \mathbb{N}^{<m}. \models P_i \Rightarrow E) \wedge \models P_m \Rightarrow E \in \text{Val} \setminus \mathbb{B} \quad \forall i \in \mathbb{N}^{<m}. \Gamma \vdash (P_i) \ C \text{ (ok : } P_{i+1}\text{) (err : } Q_i) \quad E_{err} \triangleq [\text{"Type"}, \text{str}(E), E, \text{"Bool"}]}{\Gamma \vdash (P_0) \text{ while } (E) \ C \text{ (err : } (\exists n < m. Q_n) \vee (P_m \star \text{err} = E_{err}))}
\end{array}$$

$$\begin{array}{c}
\text{EQUIV} \\
\frac{\Gamma \vdash (P') C (ok : Q'_{ok}) (err : Q'_{err}) \models P', Q'_{ok}, Q'_{err} \Leftrightarrow P, Q_{ok}, Q_{err}}{\Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err})} \\
\\
\text{FRAME} \qquad \text{EXISTS} \\
\frac{\Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err}) \quad \text{mod}(C) \cap \text{fv}(R) = \emptyset}{\Gamma \vdash (P \star R) C (ok : Q_{ok} \star R) (err : Q_{err} \star R)} \qquad \frac{\Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err})}{\Gamma \vdash (\exists x. P) C (ok : \exists x. Q_{ok}) (err : \exists x. Q_{err})} \\
\\
\text{DISJ} \\
\frac{\Gamma \vdash (P_1) C (ok : Q_{ok}^1) (err : Q_{err}^1) \quad \Gamma \vdash (P_2) C (ok : Q_{ok}^2) (err : Q_{err}^2)}{\Gamma \vdash (P_1 \vee P_2) C (ok : Q_{ok}^1 \vee Q_{ok}^2) (err : Q_{err}^1 \vee Q_{err}^2)} \\
\\
\text{FCALL} \\
\frac{(\vec{x} = \vec{x} \star P) (Q_{ok}) (Q_{err}) \in \Gamma(f) \quad y \notin \text{pv}(E_y) \quad \theta \stackrel{\text{def}}{=} [E_y/y]}{\Gamma \vdash (y = E_y \star \vec{E} = \vec{x} \star P) y := f(\vec{E}) (ok : \vec{E}\theta = \vec{x} \star Q_{ok}[y/\text{ret}]) (err : y = E_y \star \vec{E} = \vec{x} \star Q_{err})} \\
\\
\text{FCALL-ERR-FCT-NOFUNC} \qquad \text{ENV-EMPTY} \\
\frac{f \notin \text{dom}(\Gamma) \quad E_{err} \triangleq [\text{"NoFunc"}, f]}{\Gamma \vdash (y = E_y \star \vec{E} = \vec{x}) y := f(\vec{E}) (err : Q_{err})} \qquad \vdash (\emptyset, \emptyset) \\
\\
\text{ENV-EXTEND} \\
\vdash (Y, \Gamma) \quad I = \{1, \dots, n\} \quad \forall i \in I. f_i \notin \text{dom}(\gamma) \quad \gamma' = \gamma[f_i \mapsto (\vec{x}_i, C_i, E_i)]_{i \in I} \\
\Gamma(\alpha) = \Gamma[f_i \mapsto \{(P^i(\beta)) (ok : Q_{ok}^i(\beta))(err : Q_{err}^i(\beta)) \mid \beta < \alpha\} \cup \{(P_{\infty}^i(\beta)) (\text{False}) \mid \beta \leq \alpha\}]_{i \in I} \\
\forall i \in I, \alpha. \exists t \in \text{Int}_{\gamma', f_i}((P^i(\alpha)) (ok : Q_{ok}^i(\alpha))(err : Q_{err}^i(\alpha))). \Gamma(\alpha) \vdash C_i : t \\
\forall i \in I, \alpha. \exists t \in \text{Int}_{\gamma', f_i}((P_{\infty}^i(\alpha)) (\text{False})). \Gamma(\alpha) \vdash C_i : t \\
P^i \triangleq \exists \alpha. P^i(\alpha) \star \alpha \in \mathcal{O} \qquad P_{\infty}^i \triangleq \exists \alpha. P_{\infty}^i(\alpha) \star \alpha \in \mathcal{O} \\
Q_{ok}^i \triangleq \exists \alpha. Q_{ok}^i(\alpha) \star \alpha \in \mathcal{O} \qquad Q_{err}^i \triangleq \exists \alpha. Q_{err}^i(\alpha) \star \alpha \in \mathcal{O} \\
\Gamma'' := \Gamma[f_i \mapsto \{(P^i) (ok : Q_{ok}^i)(err : Q_{err}^i), (P_{\infty}^i) (\text{False})\}]_{i \in I} \\
\hline
\vdash (\gamma', \Gamma'')
\end{array}$$

C PROOF OF SOUNDNESS: ESL

In order to prove the soundness result of theorem 4.12, we require three auxiliary lemmas regarding environment validity. Their proofs are straightforward and will therefore be omitted. For legacy reasons, this Appendix and the following Appendices may use an alternative notation for satisfiability: $\theta, s, h \models P$ instead of $\theta, (s, h) \models P$, where θ (referred to as the substitution) is of type: $\text{LVar} \rightarrow_{\text{fin}} \text{Val}$.

LEMMA C.1 (AUXILIARY PROPERTIES). *The following properties hold:*

- (1) $\forall \theta, s, s', h, P. \theta, s, h \models P \wedge s|_{\text{pv}(P)} = s'|_{\text{pv}(P)} \Rightarrow \theta, s', h \models P$
- (2) $\forall \theta, s, s', h, h', o, C, \gamma. (s, h), C \Downarrow_{\gamma} o : (s', h') \Rightarrow \forall x \in \text{dom}(s) \setminus \text{mod}(C). s(x) = s'(x)$
- (3) $\forall E, \theta, s, s', \gamma. s|_{\text{pv}(E) \setminus \{\gamma\}} = s'|_{\text{pv}(E) \setminus \{\gamma\}} \Rightarrow \llbracket E[s'(y)/y] \rrbracket_{\theta, s} = \llbracket E \rrbracket_{\theta, s'}$

PROOF OF THEOREM 4.12. By induction on the derivation $\Gamma \vdash (P) \ C \ (ok : Q_{ok}) \ (err : Q_{err})$. We prove a representative selection of rules; the proofs for the remaining ones are analogous.

Function Call. We first prove the successfully terminating case of under-approximation. Our hypotheses are:

- (H1) $\models (\gamma, \Gamma)$,
- (H2) $\theta, s', h' \models \vec{E}[E_y/y] = \vec{x} \star Q_{ok}[y/\text{ret}]$,
- (H3) $h' \# h_f$,
- (H5) $\gamma \notin \text{fv}(E_y)$,
- (H6) $(\vec{x} = \vec{x} \star P) \ (ok : Q_{ok})(err : Q_{err}) \in \Gamma(f)$

Our goal is to show that:

$$\exists s, h. \theta, s, h \models \gamma = E_y \ \vec{E} = \vec{x} \star P \wedge (s, h \uplus h_f), \gamma := f(\vec{x}) \Downarrow_{\gamma} ok : (s', h' \uplus h_f)$$

- From (H6) and (H1), we obtain C and E , such that (H7) $f(\vec{x})\{C; \text{return } E\} \in \gamma$.
- From (H2), we obtain that (H2a) $\theta, s' \models \vec{E}[E_y/y] = \vec{x}$, and (H2b) $\theta, s', h' \models Q_{ok}[y/\text{ret}]$.
- From (H1), (H6) and (H7), we obtain that there exists a specification

$$(\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) \ (Q'_{ok})(Q'_{err}) \in \text{Int}_{\gamma, f}(\vec{x} = \vec{x} \star P) \ (Q_{ok})(Q_{err})$$

such that (H8) $\gamma \models (\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) \ C \ (ok : Q'_{ok}) \ (err : Q'_{err})$, where, from the definition of the internalisation function, we know that (H9a) $Q_{ok} \Leftrightarrow \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \text{ret} = E[\vec{p}/\vec{p}]$, where $\vec{z} = \text{pv}(C) \setminus \{\vec{x}\}$ and $\vec{p} = \{\vec{x}\} \uplus \{\vec{z}\} = \text{pv}(C)$.

- Given (H2b) and (H9a), we derive the following:

$$\begin{aligned} \theta, s', h' &\models (\exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \text{ret} = E[\vec{p}/\vec{p}])[y/\text{ret}] \\ &\Rightarrow \theta, s', h' \models \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \gamma = E[\vec{p}/\vec{p}] \end{aligned}$$

from which we obtain that there exist values \vec{w} , such that:

$$\begin{aligned} &\Rightarrow \theta[\vec{p} \rightarrow \vec{w}], s', h' \models Q'_{ok}[\vec{p}/\vec{p}] \star \gamma = E[\vec{p}/\vec{p}] \\ &\Rightarrow \theta[\vec{p} \rightarrow \vec{w}], s', h' \models Q'_{ok}[\vec{w}/\vec{p}] \star \gamma = E[\vec{w}/\vec{p}] \\ &\Rightarrow \theta[\vec{p} \rightarrow \vec{w}], s'[\vec{p} \rightarrow \vec{w}], h' \models Q'_{ok} \star \gamma = E \\ &\Rightarrow \theta, s'[\vec{p} \rightarrow \vec{w}], h' \models Q'_{ok} \star \gamma = E \quad (\text{H10a}) \\ &\Rightarrow \theta, s'[\vec{p} \rightarrow \vec{w}], h' \models Q'_{ok} \quad (\text{H10b}) \end{aligned}$$

- Instantiating (H8) with (H10b) and (H3), we obtain that there exist \tilde{s} and h , such that (H11) $\theta, \tilde{s}, h \models \vec{x} = \vec{x} \star P \star \vec{z} = \text{null}$ and (H12) $(\tilde{s}, h \uplus h_f), C \Downarrow_{\gamma} ok : (s'[\vec{p} \rightarrow \vec{w}], h' \uplus h_f)$.
- Let $\vec{v} = \theta(\vec{x})$. Then, since $\text{pv}(P) = \emptyset$ and given Lemma C.1(1), taking $s'' := \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}]$, we obtain that (H13) $\theta, s'', h \models \vec{x} = \vec{x} \star P \star \vec{z} = \text{null}$ and also that (H14) $(s'', h \uplus h_f), C \Downarrow_{\gamma} ok : (s'[\vec{p} \rightarrow \vec{w}], h' \uplus h_f)$.
- Let $v' = \llbracket E \rrbracket_{s'[\vec{p} \rightarrow \vec{w}]} = \llbracket E[\vec{w}/\vec{p}] \rrbracket_{\theta, s'}, v_y = \llbracket E_y \rrbracket_{\theta, s'}$ and (H15) $s = s'[y \rightarrow v_y]$. Therefore, we also have that (H16) $s' = s[y \rightarrow v']$.
- We now need to prove that $(s, h \uplus h_f), \gamma := f(\vec{E}) \Downarrow_{\gamma} ok : (s[y \rightarrow v'], h' \uplus h_f)$. For this, we already have: $f(\vec{x})\{C; \text{return } E\} \in \gamma, \text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\}, s'' = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}]$, $(s'', h \uplus h_f), C \Downarrow_{\gamma} (s', h' \uplus h_f)$ and $\llbracket E \rrbracket_{s'[\vec{p} \rightarrow \vec{w}]} = v'$, and we still need $\llbracket \vec{E} \rrbracket_s = \vec{v}$. Rewriting (H2a) given (H16), we get $\theta, s[y \rightarrow v'] \models$

$\vec{E}[E_y/y] = \vec{x}$, that is, **(H17)** $\llbracket \vec{E}[E_y/y] \rrbracket_{\theta, s[y \rightarrow v']} = \vec{v}$. From (H17), the definition of s , and Lemma C.1(3), we then obtain that $\llbracket \vec{E} \rrbracket_{\theta, s} = \vec{v}$, and from there, as \vec{E} are program expressions, we obtain the desired $\llbracket \vec{E} \rrbracket_s = \vec{v}$.

- Finally, we need to prove that $\theta, s, h \models y = E_y \star \vec{E} = \vec{x} \star P$. For the first “starjunct”, we need to show $\llbracket y \rrbracket_{\theta, s} = \llbracket E_y \rrbracket_{\theta, s}$:

$$\llbracket y \rrbracket_{\theta, s} = s(y) \stackrel{(H15)}{=} (s'[y \rightarrow v_y])(y) = v_y = \llbracket E_y \rrbracket_{\theta, s'} \stackrel{(H16)}{=} \llbracket E_y \rrbracket_{\theta, s[y \rightarrow v']} \stackrel{(H5)}{=} \llbracket E_y \rrbracket_{\theta, s}$$

For the second starjunct we do:

$$\llbracket \vec{x} \rrbracket_{\theta, s} = \theta(\vec{x}) = \llbracket \vec{x} \rrbracket_{\theta, s'} \stackrel{(H2a)}{=} \llbracket \vec{E}[E_y/y] \rrbracket_{\theta, s'} \stackrel{(H15)}{=} \llbracket \vec{E} \rrbracket_{\theta, s}$$

The third starjunct follows from (H13), given that P has no logical variables by construction.

Next, we prove the erroneously terminating case of under-approximation. Our hypotheses are:

- (H1)** $\models (\gamma, \Gamma)$,
- (H2)** $\theta, s', h' \models \gamma = E_y \star \vec{E} = \vec{x} \star Q_{err}$,
- (H3)** $h' \not\models h_f$,
- (H4)** $\gamma \notin \text{fv}(E_y)$,
- (H5)** $(\vec{x} = \vec{x} \star P) (Q_{ok})(Q_{err}) \in \Gamma(f)$.

Our goal is to show that:

$$\exists s, h. \theta, s, h \models \gamma = E_y \star \vec{E} = \vec{x} \star P \wedge (s, h \uplus h_f), \gamma := f(\vec{x}) \Downarrow_{\gamma} \text{err} : (s', h' \uplus h_f)$$

- From (H5) and (H1), we obtain C and E , such that **(H7)** $f(\vec{x})\{C; \text{return } E\} \in \gamma$.
- From (H2), we obtain that **(H2a)** $\theta, s' \models \vec{E} = \vec{x}$, **(H2b)** $\theta, s', h' \models Q_{err}$ and **(H2c)** $\theta, s' \models \gamma = E_y$.
- From (H1), (H5) and (H7), we obtain that there exists a specification

$$(\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) (Q'_{ok})(Q'_{err}) \in \text{Int}_{\gamma, f}((\vec{x} = \vec{x} \star P)(Q_{ok})(Q_{err}))$$

such that **(H8)** $\gamma \models (\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) C \left(\text{ok} : Q'_{ok} \right) (err : Q'_{err})$, where, from the definition of the internalisation function, we know that **(H9a)** $Q_{err} \Leftrightarrow \exists \vec{p}. Q'_{err}[\vec{p}/\vec{p}]$ with $\vec{z} = \text{pv}(C) \setminus \{\vec{x}\}$ and $\vec{p} = \{\vec{x}\} \uplus \{\vec{z}\}$.

- Given (H2b) and (H9a), analogous to the previous case we obtain:

$$\theta, s', h' \models (\exists \vec{p}. Q'_{err}[\vec{p}/\vec{p}]) \implies \exists \vec{w}. \theta, s'_p, h' \models Q'_{err} \quad \textbf{(H10)}$$

where $s'_p = s'[\vec{p} \mapsto \vec{w}]$

- Instantiating (H8) with (H1), (H10), and (H3), we obtain that there exist \tilde{s} and h , such that **(H11)** $\theta, \tilde{s}, h \models \vec{x} = \vec{x} \star P \star \vec{z} = \text{null}$ and **(H12)** $(\tilde{s}, h \uplus h_f), C \Downarrow_{\gamma} \text{err} : (s'_p, h' \uplus h_f)$.
- Let $\vec{v} = \theta(\vec{x})$. Then, since $\text{pv}(P) = \emptyset$ and given Lemma C.1(1), taking $s'' := \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}]$, we obtain that **(H13)** $\theta, s'', h \models \vec{x} = \vec{x} \star P \star \vec{z} = \text{null}$ and also that **(H14)** $(s'', h \uplus h_f), C \Downarrow_{\gamma} \text{err} : (s'_p, h' \uplus h_f)$.
- Let $v_{err} = \llbracket \text{err} \rrbracket_{\theta, s'}$ and **(H15)** $s = s' \setminus \text{err}$. Therefore, we also have that **(H16)** $s' = s[\text{err} \rightarrow v_{err}]$.
- We now need to prove that $(s, h \uplus h_f), \gamma := f(\vec{E}) \Downarrow_{\gamma} \text{err} : (s[\text{err} \rightarrow v_{err}], h' \uplus h_f)$. For this, we already have: $f(\vec{x})\{C; \text{return } E\} \in \gamma$, $\text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\}$, $s'' = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}]$ and $(s'', h \uplus h_f), C \Downarrow_{\gamma} \text{err} : (s', h' \uplus h'_f)$, and we still need $\llbracket \vec{E} \rrbracket_s = \vec{v}$. Rewriting (H2a) given (H16), we get $\theta, s' \models \vec{E} = \vec{x}$, that is, **(H17)** $\llbracket \vec{E} \rrbracket_{\theta, s'} = \vec{v}$. From (H16) and (H17), we then obtain that $\llbracket \vec{E} \rrbracket_{\theta, s} = \vec{v}$, which yields $\llbracket \vec{E} \rrbracket_s = \vec{v}$ since E is a program expression.
- Finally, we need to prove that $\theta, s, h \models \gamma = E_y \star \vec{E} = \vec{x} \star P$. The first starjunct is proven as follows:

$$\llbracket y \rrbracket_{\theta, s} = s(y) \stackrel{(H16)}{=} s'(y) = \llbracket E_y \rrbracket_{\theta, s'} \stackrel{(H16)}{=} \llbracket E_y \rrbracket_{\theta, s}$$

For the second starjunct we do:

$$\llbracket \vec{x} \rrbracket_{\theta, s} = \theta(\vec{x}) = \llbracket \vec{x} \rrbracket_{\theta, s'} \stackrel{(H2a)}{=} \llbracket \vec{E} \rrbracket_{\theta, s'} \stackrel{(H15)}{=} \llbracket \vec{E} \rrbracket_{\theta, s}$$

The third starjunct follows from (H13), given that P has no logical variables by construction.

We move on to proving the successfully terminating over-approximation soundness. Our hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $\theta, s, h \models \gamma = E_y \star \vec{E} = \vec{x} \star P$
- (H3) $h \# h_f$
- (H4) $\gamma \notin \text{fv}(E_y)$
- (H5) $(\vec{x} = \vec{x} \star P) (ok : Q_{ok})(err : Q_{err}) \in \Gamma(f)$.

Our goal is to show that:

$$\begin{aligned} & \forall s', h''. (s, h \uplus h_f), \gamma := f(\vec{x}) \Downarrow_{\gamma} ok : (s', h'') \\ & \Rightarrow (o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models \vec{E}[E_y/\gamma] = \vec{x} \star Q_{ok}[\gamma/\text{ret}]) \end{aligned}$$

- From (H1) and (H5), we obtain C and E such that (H6) $f(\vec{x})\{C, \text{return } E\} \in \gamma$.
- We define $\vec{v} := \theta(\vec{x})$ and obtain from (H2) that (H2a) $\theta, s \models \vec{E} = \vec{x}$ and $\theta, s, h \models P$, and hence (H2b) $\theta, s[\vec{x} \rightarrow \vec{v}], h \models \vec{x} = \vec{x} \star P$.
- Since $\text{pv}(P) = \emptyset$ we obtain from (H2b) with Lemma 1 that $\theta, \emptyset[\vec{x} \rightarrow \vec{v}], h \models \vec{x} = \vec{x} \star P$ and hence (H7) $\theta, \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}], h \models \vec{x} = \vec{x} \star P \star \vec{z} = \text{null}$.
- (H1), (H5) and (H6) imply the existence of a specification $(\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) (ok : Q'_{ok})(err : Q'_{err}) \in \text{Int}_{\gamma, f}((\vec{x} = \vec{x} \star P) (ok : Q_{ok})(err : Q_{err}))$ such that

$$(H9) \quad \gamma \models (\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) C \left(ok : Q'_{ok} \right) (err : Q'_{err})$$

- Instantiating (H9) with (H1), (H7) and (H3) yields

$$(H11) \quad \begin{aligned} & \forall s', h''. (s'', h \uplus h_f), C \Downarrow_{\gamma} ok : (s', h'') \\ & \Rightarrow (o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models Q'_{ok}) \end{aligned}$$

- Defining $v' := \llbracket E \rrbracket_{\theta, s'}$, we apply the operation semantics of the successfully termination function call, which yields

$$(s, h \uplus h_f), \gamma := f(\vec{E}) \Downarrow_{\gamma} (s[\gamma \rightarrow v'], h'')$$

- To conclude the proof, it remains to show that $\theta, s[\gamma \rightarrow v'], h' \models \vec{E}[E_y/\gamma] = \vec{x} \star Q_{ok}[\gamma/\text{ret}]$. (H11) implies $\theta, s', h' \models Q'_{ok}$. Defining $\vec{p} := \text{pv}(Q'_{ok})$ and $\vec{v} := s'(\vec{p})$, we obtain $\theta[\vec{p} \rightarrow \vec{v}], -, h' \models Q'_{ok}[\vec{p}/\vec{p}]$ where $-$ may denote any variable store, since the assertion does not hold any program variables. Therefore, $\theta[\vec{p} \rightarrow \vec{v}], s, h' \models Q'_{ok}[\vec{p}/\vec{p}]$ and hence (H12) $\theta, s[\gamma \rightarrow v'], h' \models \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \gamma = v'$ hold.
- From the definitions of v' , \vec{p} and \vec{p} we obtain $v' := \llbracket E \rrbracket_{\theta, s'} = \llbracket E[\vec{p}/\vec{p}] \rrbracket_{\theta, -}$ and therefore $\theta, s[\gamma \rightarrow v'], h' \models \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \gamma = E[\vec{p}/\vec{p}]$. Hence $\theta, s[\gamma \rightarrow v'], h' \models Q_{ok}[\gamma/\text{ret}]$.
- From (H2a) and Lemma 3, we obtain $\theta, s[\gamma \rightarrow v'] \models \vec{E}[E_y/\gamma] = \vec{x}$ and therefore $\theta, s[\gamma \rightarrow v'], h' \models \vec{E}[E_y/\gamma] = \vec{x} \star Q_{ok}[\gamma/\text{ret}]$, which concludes this case of the proof.

Finally, we prove the erroneously terminating over-approximation soundness. Our hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $\theta, s, h \models \gamma = E_y \star \vec{E} = \vec{x} \star P$
- (H3) $h \# h_f$
- (H4) $\gamma \notin \text{fv}(E_y)$
- (H5) $(\vec{x} = \vec{x} \star P) (ok : Q_{ok})(err : Q_{err}) \in \Gamma(f)$.

Our goal is to show that:

$$\begin{aligned} & \forall s', h''. (s, h \uplus h_f), \gamma := f(\vec{x}) \Downarrow_{\gamma} err : (s', h'') \\ & \Rightarrow (o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models \gamma = E_y \star \vec{E} = \vec{x} \star Q_{err}) \end{aligned}$$

- From (H1) and (H5), we obtain C and E such that (H6) $f(\vec{x})\{C, \text{return } E\} \in \gamma$.
- We define $\vec{v} := \theta(\vec{x})$ and obtain from (H2), that (H2a) $\theta, s \models \vec{E} = \vec{x}$ and $\theta, s, h \models P$, and hence (H2b) $\theta, s[\vec{x} \rightarrow \vec{v}], h \models \vec{x} = \vec{x} \star P$.

- Since $\text{pv}(P) = \emptyset$ we obtain from (H2b) with Lemma 1 that $\theta, \emptyset[\vec{x} \rightarrow \vec{v}], h \models \vec{x} = \vec{x} \star P$ and (H7) $\theta, \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}], h \models \vec{x} = \vec{x} \star P \star \vec{z} = \text{null}$.
- (H1), (H5) and (H6) imply the existence of a specification $(\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) (ok : Q'_{ok})(err : Q'_{err}) \in \text{Int}_{\gamma, f}((\vec{x} = \vec{x} \star P)(Q_{ok})(Q_{err}))$ such that

$$(H9) \quad \gamma \models (\vec{x} = \vec{x} \star P \star \vec{z} = \text{null}) C (ok : Q'_{ok}) (err : Q'_{err})$$

- Instantiating (H9) with (H1), (H7) and (H3) yields

$$(H11) \quad \begin{aligned} & \forall s', h'', (s'', h \uplus h_f), C \Downarrow_{\gamma} err : (s', h'') \\ & \Rightarrow (o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models Q'_{err}) \end{aligned}$$

- Defining $v_{err} := \llbracket err \rrbracket_{s'}$, we apply the operation semantics of the erroneously termination function call, which yields

$$(s, h \uplus h_f), \gamma := f(\vec{E}) \Downarrow_{\gamma} (s[err \rightarrow v_{err}], h'')$$

- To conclude the proof, it remains to show that $\theta, s[err \rightarrow v_{err}], h' \models \gamma = E_y \star \vec{E} = \vec{x} \star Q_{err}$. $\theta, s' \models \gamma = E_y$ holds trivially and (H11) implies $\theta, s', h' \models Q'_{err}$. Defining $\vec{p} := \text{pv}(Q'_{err})$ and $\vec{v} := s'(\vec{p})$, we obtain $\theta[\vec{p} \rightarrow \vec{v}], -, h' \models Q'_{err}[\vec{p}/\vec{p}]$ where $-$ may denote any variable store, since the assertion does not hold any program variables. Therefore, $\theta[\vec{p} \rightarrow \vec{v}], s[err \rightarrow v_{err}], h' \models Q'_{err}[\vec{p}/\vec{p}]$ and hence $\theta, s[err \rightarrow v_{err}], h' \models \exists \vec{p}. Q'_{err}[\vec{p}/\vec{p}]$ and (H12) $\theta, s[err \rightarrow v_{err}], h' \models Q_{err}$ holds.
- (H2) implies $\theta(\vec{x}) = \llbracket \vec{E} \rrbracket_{\theta, s} = \llbracket \vec{E} \rrbracket_{\theta, s[err \rightarrow v_{err}]}$ and hence $\theta, s[err \mapsto v_{err}] \models \vec{E} = \vec{x}$. Therefore, we obtain $\theta, s[err \rightarrow v_{err}], h' \models \gamma = E_y \star \vec{E} = \vec{x} \star Q_{err}$, which concludes the proof.

While. The iterative while rule is:

$$\frac{\begin{array}{l} \text{WHILE-ITERATE} \\ \forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B} \quad \forall i \in \mathbb{N}. \Gamma \vdash (P_i \wedge E) C (ok : P_{i+1}) (err : Q_i) \\ P_{\infty} \triangleq \text{False} \quad m \triangleq \min(\{i \in \mathbb{N} \cup \{\infty\} \mid \models P_i \Rightarrow \neg E\}) \end{array}}{\Gamma \vdash (P_0) \text{ while } (E) C (ok : P_m) (err : \exists n < m. Q_n)}$$

We prove the under-approximation case for successful termination; the faulting case is proven analogously.

Our hypotheses are as follows:

- (H1) $\models (\gamma, \Gamma)$;
- (H2a) $\forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B}$
- (H2b) $\forall i \in \mathbb{N}. \Gamma \vdash (P_i \wedge E) C (ok : P_{i+1}) (err : Q_i)$;
- (H3) $P_{\infty} \triangleq \text{False}$;
- (H4) $m \triangleq \min(\{i \in \mathbb{N} \cup \{\infty\} \mid \models P_i \Rightarrow \neg E\})$;
- (H5) $\theta, s', h' \models P_m$;
- (H6) $h' \not\models h_f$.

Our goal is to show that:

$$\exists s, h. \theta, s, h \models P_0 \wedge (s, h \uplus h_f), \text{while } (E) C \Downarrow_{\gamma} ok : (s', h' \uplus h_f)$$

- From (H5), we have that (H7) $m \neq \infty$;
- If $m = 0$, we have from (H4) and (H5) that $\theta, s', h' \models P_0 \wedge \neg E$. Taking $s = s'$ and $h = h'$, we have $\llbracket E \rrbracket_s = \text{false}$, and the operational semantics yields the required $(s, h \uplus h_f), \text{while } (E) C \Downarrow_{\gamma} (s', h' \uplus h_f)$;
- Otherwise, we have that $m > 0$, and (H4) and (H5) imply that (H8) $\theta, s', h' \models P_m$. Then, by iterative application of (H2a), (H2b), and the induction hypothesis, we obtain the existence of a state s, h such that (H9) $(\theta, s, h \models P_0 \wedge E)$ and (H10) $(s, h \uplus h_f), C^m \Downarrow_{\gamma} (s', h' \uplus h_f)$. From (H9), it also follows that $\theta, s, h \models P_0$. Finally, given (H4), (H10), and the operational semantics of the while loop, we also have that $\text{while } (E) C \Downarrow_{\gamma} ok : (s', h' \uplus h_f)$.

Second, we prove the over-approximating case. Our hypotheses are as follows:

- (H1) $\models (\gamma, \Gamma)$;
- (H2a) $\forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B}$
- (H2b) $\forall i \in \mathbb{N}. \Gamma \vdash (P_i \wedge E) C (ok : P_{i+1}) (err : Q_i)$;

- (H3) $P_\infty \triangleq \text{False}$;
 (H4) $m \triangleq \min(\{i \in \mathbb{N} \cup \{\infty\} \mid \models P_i \Rightarrow \neg E\})$;
 (H5) $\theta, s, h \models P_0 \star E \in \mathbb{B}$,
 (H6) $h' \# h_f$.

and our goal is to show that:

$$\begin{aligned} & \forall o, s', h''. (s, h \uplus h_f), \text{while } (E) C \Downarrow_Y o : (s', h'') \implies \\ & \exists h'. h'' = h' \uplus h_f \wedge ((o = \text{ok} \wedge \theta, s', h' \models P_m) \vee (o = \text{err} \wedge \theta, s', h' \models \exists n < m. Q_n)) \end{aligned}$$

Taking (H7) $(s, h \uplus h_f), \text{while } (E) C \Downarrow_Y o : (s', h'')$, we have the following:

- If $m = 0$, then $\theta, s, h \models P_0 \wedge \neg E$, meaning that $\llbracket E \rrbracket_s = \text{false}$, and from there, the operational semantics yields $(s, h \uplus h_f), \text{while } (E) C \Downarrow_Y (s, h \uplus h_f)$ and, trivially, it also holds that $\theta, s, h \models P_m$.
- If (H8) $m = \infty$, then given (H7) we can prove by contradiction that $o = \text{err}$. Then, let $k > 0$ be the number of times the while unrolling rule has been applied in the derivation of (H7) (such a k exists due to the design of the operational semantics). Then, given (H2), (H4), (H8), the inductive hypothesis, and the semantics of sequencing, we have that

$$\begin{aligned} \text{(H9a)} \quad & \forall o, s', h''. (s, h \uplus h_f), C^k \Downarrow_Y o : (s', h'') \implies \\ & \exists h'. h'' = h' \uplus h_f \wedge ((o = \text{ok} \wedge \theta, s', h' \models P_k) \vee (o = \text{err} \wedge \theta, s', h' \models Q_{k-1})) \end{aligned}$$

and since the final states (s', h'') coincide for (H9a) and (H7) given the operational semantics of the while loop, we have the desired goal for $n = k - 1 < m = \infty$.

- Otherwise, we have that $0 < m < \infty$. Then, given (H2a), (H2b), (H4), (H8), the inductive hypothesis, and the semantics of sequencing, we have that

$$\begin{aligned} \text{(H9b)} \quad & \forall o, s', h''. (s, h \uplus h_f), C^m \Downarrow_Y o : (s', h'') \implies \\ & \exists h'. h'' = h' \uplus h_f \wedge ((o = \text{ok} \wedge \theta, s', h' \models P_m) \vee (o = \text{err} \wedge \theta, s', h' \models \bigvee_{i=0}^{m-1} Q_i)) \end{aligned}$$

and since the final states (s', h'') coincide for (H9b) and (H7) given the operational semantics of the while loop, we have the desired goal, where $n < m$ is guaranteed by the bounds of the disjunction.

Frame. The frame rule is:

$$\frac{\text{FRAME} \quad \Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err}) \quad \text{mod}(C) \cap \text{fv}(R) = \emptyset}{\Gamma \vdash (P \star R) C (ok : Q_{ok} \star R) (err : Q_{err} \star R)}$$

To prove the soundness of this rule, our hypotheses are that for arbitrary γ :

- (H1) $\models (\gamma, \Gamma)$
 (H2) $\Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err})$
 (H3) $\text{mod}(C) \cap \text{fv}(R) = \emptyset$

From the inductive hypothesis and (H2), it follows that $\Gamma \models (P) C (ok : Q_{ok}) (err : Q_{err})$ (H4). It then suffices to show that $\Gamma \models (P \star R) C (ok : Q_{ok} \star R) (err : Q_{err} \star R)$ holds. We start off by showing the over-approximating case. For this we assume that for some $\theta, s, h, h_f, o, s', h''$:

- (H5) $\theta, s, h \models P \star R$
 (H6) $(s, h \uplus h_f), C \Downarrow_Y o : (s', h'')$

From the definition of the satisfiability relation and (H5), it follows that there exists some heaps, \bar{h} and \bar{h}_r , such that:

- (H7) $h = \bar{h} \uplus \bar{h}_r$
 (H8) $\theta, s, \bar{h} \models P$
 (H9) $\theta, s, \bar{h}_r \models R$

Letting $\bar{h}_f = \bar{h}_r \uplus h_f$, from (H6) and the associativity of \uplus , it follows that $(s, \bar{h} \uplus \bar{h}_f), C \Downarrow_Y o : (s', h'')$ (H10). From (H4), (H1), (H8) and (H10), it follows that:

$$o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus \bar{h}_f \wedge \theta, s', h' \models Q_o$$

By applying item 2 and item 1 to (H9), we can infer that $\theta, s', \bar{h}_r \models R$. Then letting $\bar{h}' = h' \uplus \bar{h}_r$, given the definition of the satisfiability relation, it follows that $\theta, s', \bar{h}' \models Q_o \star R$. We can then infer that:

$$o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus \bar{h}_f \wedge \theta, s', h' \models Q_o \star R$$

as required. We now show the under-approximating case. For this we assume that for some θ, s', h', h_f, o :

$$(H11) \quad \theta, s', h' \models Q_o \star R$$

$$(H12) \quad h_f \# h'$$

From the definition of the satisfiability relation and (H11), it follows that there exists some heaps, \bar{h}' and \bar{h}'_r such that:

$$(H13) \quad h' = \bar{h}' \uplus \bar{h}'_r$$

$$(H14) \quad \theta, s', \bar{h}' \models Q_o$$

$$(H15) \quad \theta, s', \bar{h}'_r \models R$$

Letting $\bar{h}_f = h_f \uplus \bar{h}'_r$, from (H12) and (H13), it follows that $\bar{h}_f \# \bar{h}'$ (H16). From (H4), applying (H1), (H14) and (H16), it follows that:

$$\exists s, \bar{h}. \theta, s, \bar{h} \models P \wedge (s, \bar{h} \uplus \bar{h}_f), C \Downarrow_Y o : (s', \bar{h}' \uplus \bar{h}_f)$$

By applying item 2 and item 1 to (H15), it follows that $\theta, s, \bar{h}'_r \models R$. Letting $h = \bar{h} \uplus \bar{h}'_r$, by the definition of the satisfiability relation, it follows that:

$$\exists s, \bar{h}. \theta, s, h \models P \star R \wedge (s, h \uplus h_f), C \Downarrow_Y o : (s', h' \uplus h_f)$$

as required.

Equivalence. The equivalence rule is

$$\frac{\text{EQUIV} \quad \Gamma \vdash (P') \ C \ (ok : Q'_{ok}) \ (err : Q'_{err}) \quad \models P', Q'_{ok}, Q'_{err} \Leftrightarrow P, Q_{ok}, Q_{err}}{\Gamma \vdash (P) \ C \ (ok : Q_{ok}) \ (err : Q_{err})}$$

Our hypotheses for OX-soundness are

$$(H1) \quad \models (\gamma, \Gamma)$$

$$(H2) \quad \Gamma \vdash (P') \ C \ (ok : Q'_{ok}) \ (err : Q'_{err})$$

$$(H3) \quad \models P', Q'_{ok}, Q'_{err} \Leftrightarrow P, Q_{ok}, Q_{err}$$

$$(H4) \quad \theta, s, h \models P$$

$$(H5) \quad (s, h \uplus h_f), C \Downarrow_Y (s', h'')$$

and we aim to show that

$$o \neq \text{miss} \wedge \exists h'. h = h' \uplus h_f \wedge \theta, s', h' \models Q_o$$

(H3) and (H4) implies $\theta, s, h \models P'$ and with (H2) and (H5) then implies (H6) $o \neq \text{miss} \wedge \exists h'. h = h' \uplus h_f \wedge \theta, s', h' \models Q'_o$. (H2) then implies the desired result.

For the UX-soundness, the hypotheses are

$$(H1) \quad \models (\gamma, \Gamma)$$

$$(H2) \quad \Gamma \vdash (P') \ C \ (ok : Q'_{ok}) \ (err : Q'_{err})$$

$$(H3) \quad \models P', Q'_{ok}, Q'_{err} \Leftrightarrow P, Q_{ok}, Q_{err}$$

$$(H4) \quad \theta, s', h' \models Q_o$$

$$(H5) \quad h' \# h_f$$

(H2) and (H4) implies $\theta, s', h' \models Q'_o$. (H2) and (H5) implies $\exists s, h. \theta, s, h \models P' \wedge (s, h \uplus h_f), C \Downarrow_Y (s', h' \uplus h_f)$. (H3) then implies the desired result.

Existentials. The existential rule is:

$$\frac{\text{EXISTS} \quad \Gamma \vdash (P) \ C \ (ok : Q_{ok}) \ (err : Q_{err})}{\Gamma \vdash (\exists x. P) \ C \ (ok : \exists x. Q_{ok}) \ (err : \exists x. Q_{err})}$$

To prove the soundness of this rule, our hypotheses are that for arbitrary γ :

- (H1) $\models (\gamma, \Gamma)$
 (H2) $\Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err})$

Using the inductive hypothesis and (H2), it follows that (H3) $\Gamma \models (P) C (ok : Q_{ok}) (err : Q_{err})$. It then suffices to show that $\Gamma \models (\exists x. P) C (ok : \exists x. Q_{ok}) (err : \exists x. Q_{err})$. We start off by showing the over-approximating case. To do so, we assume that for some $\theta, s, h, h_f, o, s', h''$:

- (H4) $\theta, s, h \models \exists x. P$
 (H5) $(s, h \uplus h_f), C \Downarrow_Y o : (s', h' \uplus h_f)$

From (H4) and the definition of the satisfiability relation, it follows that, for some v , (H6) $\theta[x \mapsto v], s, h \models P$ holds. From (H3), (H1), (H6) and (H5), it follows that:

$$o \neq miss \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta[x \mapsto v], s', h' \models Q_o$$

This trivially entails:

$$o \neq miss \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models \exists x. Q_o$$

as required. We now show the under-approximating case. To do so, we assume that for some θ, s', h', h_f, o :

- (H7) $\theta, s', h' \models \exists x. Q_o$
 (H8) $h_f \# h'$

From (H7) and the definition of the satisfiability relation, it follows that, for some v , (H9) $\theta[x \mapsto v], s', h' \models Q_o$. From (H3), (H1), (H9) and (H8), it follows that:

$$\exists s, h. \theta[x \mapsto v], s, h \models P \wedge (s, h \uplus h_f), C \Downarrow_Y o : (s', h' \uplus h_f)$$

and consequently:

$$\exists s, h. \theta, s, h \models \exists x. P \wedge (s, h \uplus h_f), C \Downarrow_Y o : (s', h' \uplus h_f)$$

as required.

Disjunction. The disjunction rule is

$$\frac{\text{DISJ} \quad \Gamma \vdash (P_1) C (ok : Q_{ok}^1) (err : Q_{err}^1) \quad \Gamma \vdash (P_2) C (ok : Q_{ok}^2) (err : Q_{err}^2)}{\Gamma \vdash (P_1 \vee P_2) C (ok : Q_{ok}^1 \vee Q_{ok}^2) (err : Q_{err}^1 \vee Q_{err}^2)}$$

Our hypotheses for OX-soundness are

- (H1) $\models (\gamma, \Gamma)$
 (H2) $\Gamma \models (P_1) C (ok : Q_{ok}^1) (err : Q_{err}^1)$
 (H3) $\Gamma \models (P_2) C (ok : Q_{ok}^2) (err : Q_{err}^2)$
 (H4) $\theta, s, h \models P_1 \vee P_2$
 (H5) $(s, h \uplus h_f), C \Downarrow_Y o : (s', h'')$

(H4) implies that $(\theta, s, h \models P_1) \vee (\theta, s, h \models P_2)$. If the first case of the disjunct holds, (H2) implies (H6a) $o \neq miss \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models Q_o^1$. Otherwise, the second case holds and (H3) yields (H6b) $o \neq miss \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models Q_o^2$. The disjunction of (H6a) and (H6b) yields the desired result.

For the UX-soundness, our hypotheses are

- (H1) $\models (\gamma, \Gamma)$
 (H2) $\Gamma \models (P_1) C (ok : Q_{ok}^1) (err : Q_{err}^1)$
 (H3) $\Gamma \models (P_2) C (ok : Q_{ok}^2) (err : Q_{err}^2)$
 (H4) $\theta, s', h' \models Q_o^1 \vee Q_o^2$
 (H5) $h' \# h_f$

(H4) implies $(\theta, s', h' \models Q_o^1) \vee (\theta, s', h' \models Q_o^2)$. If the first case of the disjunct holds, (H2) yields (H6a) $\exists s, h(s, h \uplus h_f), C \Downarrow_Y (s', h' \uplus h_f) \wedge \theta, s, h \models P_1$. Otherwise, the second case of the disjunction holds and (H3) implies (H6b) $\exists s, h(s, h \uplus h_f), C \Downarrow_Y (s', h' \uplus h_f) \wedge \theta, s, h \models P_2$. The disjunction of (H6a) and (H6b) yields the desired result.

Lookup. The Lookup rule is

$$\text{LOOKUP} \quad \frac{x \notin \text{pv}(E') \quad \theta \triangleq [E'/x]}{\Gamma \vdash (x = E' \star E \mapsto E_1) \ x := [E] \ (E' \in \text{Val} \star x = E_1 \theta \star E \theta \mapsto E_1 \theta)}$$

To prove its OX-soundness, the hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $x \notin \text{pv}(E')$
- (H4) $\theta, s, h \models x = E' \star E \mapsto E_1$
- (H5) $(s, h \uplus h_f), x := [E] \Downarrow_Y o : (s', h'')$

and we aim to show

$$o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models E' \in \text{Val} \star x = E_1[E'/x] \star E[E'/x] \mapsto E_1[E'/x]$$

(H5) yields (H6) $o \neq \text{miss}$, (H7) $\llbracket E \rrbracket_s = v$ and (H8) $s' = s[x \rightarrow v]$ and (H9) $h'' = h \uplus h_f$. Chosing (10) $h' = h$, (H4) and (H8) yield (H11) $\theta, s' \models E' \in \text{Val}$. (H4), (H9), (H10) and (H11) imply the desired result.

For UX-soundness, our hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $x \notin \text{pv}(E')$
- (H3) $\theta, s', h' \models E' \in \text{Val} \star x = E_1[E'/x] \star E[E'/x] \mapsto E_1[E'/x]$
- (H4) $h' \# h_f$

and we aim to show:

$$\exists s, h. (s, h \uplus h_f), x := [E] \Downarrow_Y (s', h' \uplus h_f) \wedge \theta, s, h \models x = E' \star E \mapsto E_1$$

Letting $v = \llbracket E' \rrbracket_{\theta, s'}$, $s = s'[x \mapsto v]$ and $h = h'$, then:

$$(s, h \uplus h_f), x := [E] \Downarrow_Y (s', h' \uplus h_f)$$

holds. Then given (H2), by applying item 3, it is clear that $v = \llbracket E' \rrbracket_{\theta, s}$, and therefore $\theta, s \models x = E'$. Finally, similarly, $\theta, s, h \models E \mapsto E_1$, from which we can reach our goal by the definition of the satisfiability relation.

Lookup-err-val. The Lookup-err-val rule is

$$\text{LOOKUP-ERR-VAL} \quad \frac{E_{\text{err}} \triangleq [\text{"ExprEval"}, \text{str}(E)]}{\Gamma \vdash (x = E' \star E \notin \text{Val}) \ x := [E] \ (\text{err} : Q_{\text{err}})}$$

where $Q_{\text{err}} = x = E' \star E \notin \text{Val} \star \text{err} = E_{\text{err}}$. To prove OX-soundness, the hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $\theta, s, h \models x = E' \star E \notin \text{Val}$
- (H3) $(s, h \uplus h_f), x := [E] \Downarrow_Y o : (s', h'')$

It then suffices to show that $\theta, s', h'' \models x = E' \star E \notin \text{Val} \star \text{err} = E_{\text{err}}$. Given (H2) and the definition satisfiability relation, it follows that $\llbracket E \rrbracket_{s, h} \notin \text{Val}$, and therefore $\llbracket E \rrbracket_{s, h} = \downarrow$. From this we can infer that the only rule from the big-step operational semantics that can apply is:

$$\frac{\llbracket E \rrbracket_s = \downarrow \quad v_{\text{err}} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), x := [E] \Downarrow_Y \text{err} : (s_{\text{err}}, h)}$$

From this, we can infer that $h'' = h$ and $s' = s[\text{err} \mapsto v_{\text{err}}]$. From (H2) and the definition of the satisfiability relation, it then follows that $\theta, s', h'' \models Q_{\text{err}}$ as required.

For UX-soundness, our hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $\theta, s', h' \models Q_{\text{err}}$
- (H3) $h_f \# h'$

It then suffices to show that for some s, h :

$$\theta, s, h \models x = E' \star E \notin \text{Val} \wedge (s, h \uplus h_f), C \Downarrow_{\gamma} o : (s', h' \uplus h_f)))$$

Letting $s = s' \setminus \{\text{err}\}$ and $h = h'$, from (H2) and the definition of the satisfiability relation, it follows that $\theta, s, h \models x = E' \star E \notin \text{Val}$ and by applying the same big-step semantics rule as in the OX case, we derive the second starjunct of our goal as required.

Lookup-err-use-after-free. The Lookup-err-use-after-free rule is

$$\frac{\text{LOOKUP-ERR-USE-AFTER-FREE} \quad E_{\text{err}} \triangleq [\text{"UseAfterFree"}, \text{str}(E), E]}{\Gamma \vdash (x = E' \star E \mapsto \emptyset) \ x := [E] \ (err : Q_{\text{err}})}$$

where $Q_{\text{err}} = x = E' \star E \mapsto \emptyset \star \text{err} = E_{\text{err}}$. To prove OX-soundness, the hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $\theta, s, h \models x = E' \star E \mapsto \emptyset$
- (H3) $(s, h \uplus h_f), x := [E] \Downarrow_{\gamma} o : (s', h'')$

It then suffices to show that $\theta, s', h'' \models x = E' \star E \mapsto \emptyset \star \text{err} = E_{\text{err}}$. Given (H2), we can infer that $h(\llbracket E \rrbracket_{s, h}) = \emptyset$. From this we can infer that the only rule from the big-step operational semantics that can apply is:

$$\frac{\llbracket E \rrbracket_s = n \quad h(n) = \emptyset \quad v_{\text{err}} = [\text{"UseAfterFree"}, \text{str}(E), n]}{(s, h), x := [E] \Downarrow_{\gamma} err : (s_{\text{err}}, h)}$$

From this, we can infer that $h'' = h$ and $s' = s[\text{err} \mapsto v_{\text{err}}]$. From (H2) and the definition of the satisfiability relation, it then follows that $\theta, s', h'' \models Q_{\text{err}}$ as required. For UX-soundness, our hypotheses are:

- (H1) $\models (\gamma, \Gamma)$
- (H2) $\theta, s', h' \models Q_{\text{err}}$
- (H3) $h_f \# h'$

It then suffices to show that for some s, h :

$$\theta, s, h \models x = E' \star E \mapsto \emptyset \wedge (s, h \uplus h_f), C \Downarrow_{\gamma} o : (s', h' \uplus h_f)))$$

Letting $s = s' \setminus \{\text{err}\}$ and $h = h'$, from (H2) and the definition of the satisfiability relation, it follows that $\theta, s, h \models x = E' \star E \mapsto \emptyset$ and by applying the same big-step semantics rule as in the OX case, we derive the second conjunct of our goal as required.

New. The New rule is

$$\frac{\text{NEW} \quad x \notin \text{pv}(E') \quad \theta \triangleq [E'/x]}{\Gamma \vdash (x = E' \star E \in \mathbb{N}) \ x := \text{new}(E) \ (ok : E' \in \text{Val} \star \bigotimes_{0 \leq i < E\theta} ((x+i) \mapsto \text{null}))}$$

For the OX-soundness, our hypotheses are

- (H1) $\models (\gamma, \Gamma)$
- (H2) $x \notin \text{pv}(E')$
- (H3) $\Gamma \vdash (x = E' \star E \in \mathbb{N}) \ x := \text{new}(E) \ (ok : E' \in \text{Val} \star \bigotimes_{0 \leq i < E\theta} ((x+i) \mapsto \text{null}))$
- (H4) $\theta, s, h \models x = E' \star E \in \mathbb{N}$
- (H5) $(s, h \uplus h_f), x := \text{new}(E) \Downarrow_{\gamma} o : (s', h'')$

(H5) implies:

- (H6) $o \neq \text{miss}$
- (H7) $\llbracket E \rrbracket_s = n$
- (H8) $\forall i \in \{0, \dots, n-1\}. n' + i \notin \text{dom}(h \uplus h_f)$
- (H9) $s' = s[x \mapsto n']$
- (H10) $h'' = (h \uplus h_f)[n' \mapsto \text{null}] \dots [n' + n - 1 \mapsto \text{null}]$

Defining **(H11)** $h' = h[n' \mapsto \text{null}] \dots [n' + n - 1 \mapsto \text{null}]$ yields with (H10) that **(H12)** $h'' = h' \uplus h_f$. (H2), (H4) and (H9) implies **(H13)** $\theta, s' \models E' \in \text{Val} \star x = n'$ and (H11) then implies $\theta, s', h' \models E' \in \text{Val} \star \bigotimes_{0 \leq i < E[E'/x]} (x + i) \mapsto \text{null}$, which is the desired result.

For the UX direction, the hypotheses are

- (H1)** $\models (\gamma, \Gamma)$
- (H2)** $x \notin \text{pv}(E')$
- (H3)** $\Gamma \vdash (x = E' \star E \in \mathbb{N}) \ x := \text{new}(E) \ (ok : E' \in \text{Val} \star \bigotimes_{0 \leq i < E[E'/x]} ((x + i) \mapsto \text{null}))$
- (H4)** $\theta, s', h' \models E' \in \text{Val} \star \bigotimes_{0 \leq i < E[E'/x]} ((x + i) \mapsto \text{null})$
- (H5)** $h' \# h_f$

(H4) implies that **(H5)** $x \in \text{dom}(s')$ and we define

- (H6)** $n' = s'(x)$
- (H7)** $n = \llbracket E[E'/x] \rrbracket_{\theta, s'}$
- (H8)** $h = h'|_d$, where $d = \text{dom}(h') \setminus \{s(x), \dots, s(x + n)\}$
- (H9)** $s = s'[x \mapsto v]$ where $v = \llbracket E' \rrbracket_{\theta, s'}$

(H5) and (H8) imply that **(H10)** $n' + i \notin \text{dom}(h \uplus h_f) \forall i \in \{0, \dots, n - 1\}$ and **(H11)** $h' \uplus h_f = (h \uplus h_f)[n' \mapsto \text{null}] \dots [n' + n - 1 \mapsto \text{null}]$ and (H9) implies **(H12)** $s' = s[x \mapsto n']$. (H2), (H7) and (H9) imply **(H13)** $\llbracket E \rrbracket_{\theta, s} = n$. (H10), (H11), (H12) and (H13) imply

$$(s, h \uplus h_f), x := \text{new}(E) \Downarrow_{\gamma} o : (s', h'')$$

(H2) and (H9) imply $\llbracket E \rrbracket_s = v$, which with (H7) and (H13) implies $\theta, s \models x = E' \star E \in \mathbb{N}$.

(H9), (H10), (H11) and (H13) imply the desired result.

Free. The free rule is

$$\begin{array}{c} \text{FREE} \\ \Gamma \vdash (E \mapsto E') \text{ free}(E) \ (ok : E' \in \text{Val} \star E \mapsto \emptyset) \end{array}$$

For OX-Soundness, the hypotheses are

- (H1)** $\models (\gamma, \Gamma)$
- (H2)** $\Gamma \vdash (E \mapsto E') \text{ free}(E) \ (ok : E' \in \text{Val} \star E \mapsto \emptyset)$
- (H3)** $\theta, s, h \models E \mapsto E'$
- (H4)** $(s, h \uplus h_f), \text{free}(E) \Downarrow_{\gamma} (s', h'')$

(H4) implies

- (H5)** $\llbracket E \rrbracket_s = n$
- (H6)** $(h \uplus h_f)(n) \in \text{Val}$
- (H7)** $s = s'$
- (H8)** $h'' = (h \uplus h_f)[n \mapsto \emptyset]$

(H4) and (H5) imply that $n \in \text{dom}(h)$, which with (H8) implies that $h'' = h[n \mapsto \emptyset] \uplus h_f$. (H3) and (H7) imply $\theta, s' \models E' \in \text{Val}$. (H5) and (H7) imply $\llbracket E \rrbracket_{s'} = n$ and defining $h' = h[n \mapsto \emptyset]$, we obtain $\theta, s', h' \models E' \in \text{Val} \star E \mapsto \emptyset$, which is the desired result.

For the UX direction, our hypotheses are

- (H1)** $\models (\gamma, \Gamma)$
- (H2)** $\Gamma \vdash (E \mapsto E') \text{ free}(E) \ (ok : E' \in \text{Val} \star E \mapsto \emptyset)$
- (H3)** $\theta, s', h' \models E' \in \text{Val} \star E \mapsto \emptyset$
- (H4)** $h' \# h_f$

Defining $s = s'$, (H3) yields that $n = \llbracket E \rrbracket_{\theta, s'} = \llbracket E \rrbracket_{\theta, s}$ for some $n \in \mathbb{N}$. Defining $h = h'[n \mapsto v]$ for $v = \llbracket E' \rrbracket_{\theta, s'} = \llbracket E' \rrbracket_{\theta, s}$, we obtain $h' = h[n \mapsto \emptyset]$ and therefore $h' \uplus h_f = (h \uplus h_f)[n \mapsto \emptyset]$. The operational semantics of free then yields

$$(s, h \uplus h_f), \text{free}(E) \Downarrow_{\gamma} (s', h' \uplus h_f)$$

and also obtain $\theta, s, h \models E \mapsto E'$, which is the desired result. \square

D BASICS OF SCOTT INDUCTION

The second soundness statement that needs to be proven for ESL is that well-formed environments are valid. This requires reasoning about the use of function specifications in the context of the environment extension rule.

In particular, the use of specifications of non-recursive functions is trivially sound. For recursive functions that always terminate, soundness can be proven by transfinite induction, while establishing a measure on the function pre-conditions and allowing recursive use of specifications only if they have a strictly lower measure. Without this requirement, we could prove an unsound specification (**emp**) $f()$ (**ok** : **ret** = 42) for the function $f() \{x := f(); \text{return } x\}$, which does not hold since f never terminates and the (satisfiable) post-condition **ret** = 42 implies the existence of at least one terminating execution. This soundness issue does not arise in over-approximating logics, since, due to the meaning of triples, a satisfiable post-condition does not imply the existence of terminating traces. In these logics, it is always sound to apply a specification to prove itself.

However, for recursive functions with non-terminating branches due to infinite recursion, we also have to be able to allow recursive use of specifications whose measure does not decrease, and the tool to handle such use is a form of fixpoint induction called Scott induction [45], which would normally be the tool for also proving soundness of well-formed environments in over-approximating logics. However, we were not able to find a corresponding soundness proof in the literature.

In the following, we give the relevant Scott-induction-related definitions (from [45]), together with an instantiation that will be applied to prove soundness of well-formed environments in Appendix E.

Definition D.1 (Domain). A partially ordered set (D, \sqsubseteq) is a *domain*, iff

- (D1) $\exists \perp \in D. \forall d \in D. \perp \sqsubseteq d$ (least element)
- (D2) $\forall (d_n)_{n \in \mathbb{N}} \subseteq D. (\forall i \in \mathbb{N}. d_i \sqsubseteq d_{i+1}) \implies \bigsqcup_{n \in \mathbb{N}} d_n \in D$ (chain-closedness)

where $\bigsqcup_{n \in \mathbb{N}} d_n$ denotes the least upper bound or the supremum of the set $\{d_n \mid n \in \mathbb{N}\}$ with respect to \sqsubseteq .

Definition D.2 (Admissible Subset). Given a domain (D, \sqsubseteq) with least element \perp , a subset $s \subseteq D$ is called *admissible*, iff

- (S1) $\perp \in s$ (least element)
- (S2) $\forall (s_n)_{n \in \mathbb{N}} \subseteq s. (\forall i \in \mathbb{N}. s_i \sqsubseteq s_{i+1}) \implies \bigsqcup_{n \in \mathbb{N}} s_n \in s$ (chain-closedness)

Definition D.3 (Continuity on Domains). Assuming two domains (D, \sqsubseteq_D) and (E, \sqsubseteq_E) , a function $g : D \longrightarrow E$ is *continuous*, iff

- (C1) $\forall d, d' \in D. d \sqsubseteq_D d' \implies g(d) \sqsubseteq_E g(d')$ (monotonicity)
- (C2) $\forall (d_n)_{n \in \mathbb{N}}. (\forall i \in \mathbb{N}. d_i \sqsubseteq d_{i+1}) \implies \bigsqcup_{n \in \mathbb{N}} g(d_n) = g(\bigsqcup_{n \in \mathbb{N}} d_n)$ (supremum-preservation)

THEOREM D.4 (LEAST FIXPOINT). *Given a domain D and a continuous function $g : D \longrightarrow D$, the least fixpoint of g , denoted by $\text{lfp}(g)$, has the identity*

$$\text{lfp}(g) = \bigsqcup_{n \in \mathbb{N}} g^n(\perp),$$

where \perp denotes the least element of D and g^n denotes the n -times application of g .

THEOREM D.5 (SCOTT INDUCTION PRINCIPLE). *Given a domain D , an admissible subset $s \subseteq D$, and a continuous function $g : D \longrightarrow D$, it holds that*

$$g(s) \subseteq s \implies \text{lfp}(g) \in s$$

Before presenting the instantiation of the Scott induction, we require a pseudo-command scope which models the function call, and pseudo-commands for non-deterministic choice.

Definition D.6 (The scope pseudo-command). We define a pseudo-command which closely models the behaviour of a function call:

$$\text{scope}((\vec{x}, \vec{E}), C, (y, E'))$$

whose arguments are

- a pair (\vec{x}, \vec{E}) consisting of a list of distinct program variables and a list of expressions, such that both are of the same length,
- a command C which is to be executed within the "scope",
- a tuple (y, E') of a program variable and an expression,

and whose semantics (eliding expression-evaluation fault cases) is given by

$$\frac{\begin{array}{c} \llbracket \vec{E} \rrbracket_s = \vec{v} \quad \text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\} \\ s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \quad (s_p, h), C \Downarrow_Y (s_q, h') \quad \llbracket E' \rrbracket_{s_q} = v' \end{array}}{(s, h), \text{scope}((\vec{x}, \vec{E}), C, (y, E')) \Downarrow_Y (s[y \rightarrow v'], h')}$$

$$\frac{\begin{array}{c} \llbracket \vec{E} \rrbracket_s = \vec{v} \quad \text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\} \\ s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \quad (s_p, h), C \Downarrow_Y \text{err} : (s_q, h') \quad \llbracket \text{err} \rrbracket_{s_q} = v_{\text{err}} \end{array}}{(s, h), \text{scope}((\vec{x}, \vec{E}), C, (y, E')) \Downarrow_Y \text{err} : (s[\text{err} \rightarrow v_{\text{err}}], h')}$$

Definition D.7 (Non-Deterministic Choice (pseudo-commands)). We furthermore add pseudo-commands that arbitrarily pick a command from a given set and executes it:

$$\frac{\sigma, C_1 \Downarrow_Y o : \sigma' \vee \sigma, C_2 \Downarrow_Y o : \sigma'}{\sigma, C_1 \sqcup C_2 \Downarrow_Y o : \sigma'} \quad \frac{\exists m \in \mathbb{N}. \sigma, C_m \Downarrow_Y o : \sigma'}{\sigma, \sqcup(C_n \mid n \in \mathbb{N}) \Downarrow_Y o : \sigma'}$$

We will now give some general definitions and lemmas, which will later on be instantiated to prove the soundness of the environment extension via Theorem D.5.

Definition D.8 (Greatest-Fixpoint Closure of Cmd). We define the greatest-fixpoint closure of the set of commands and pseudo-commands, $\text{Cmd} \cup \{\text{scope}, \sqcup, \sqcup\}$, as the closure of that set under infinite applications of the command constructors, and denote that closure by \mathbb{C} .

Definition D.9 (Behavioural Equivalence on \mathbb{C}). Given an arbitrary function implementation context γ , we define the equivalence relation \approx_γ on \mathbb{C} as

$$C_1 \approx_\gamma C_2 \iff \{(\sigma, \sigma') \in \text{State}^2 \mid \exists o. \sigma, C_1 \Downarrow_Y o : \sigma'\} = \{(\sigma, \sigma') \in \text{State}^2 \mid \exists o. \sigma, C_2 \Downarrow_Y o : \sigma'\}$$

where $C_1, C_2 \in \mathbb{C}$, effectively meaning that \approx_γ relates commands that exhibit the same set of behaviours. We denote the resulting quotient space as \mathbb{C}_γ and the corresponding equivalence class of a command C by $[C]$. This relation yields a partial order, denoted by \sqsubseteq_γ and defined as:

$$C_1 \sqsubseteq_\gamma C_2 \iff \{(\sigma, \sigma') \in \text{State}^2 \mid \exists o. \sigma, C_1 \Downarrow_Y o : \sigma'\} \subseteq \{(\sigma, \sigma') \in \text{State}^2 \mid \exists o. \sigma, C_2 \Downarrow_Y o : \sigma'\}$$

Furthermore, we define the join operator on commands in $\mathbb{C}_\gamma, \sqcup_\gamma$, as the non-deterministic choice, lift it to quotient space, overloading notation:

$$[C_1] \sqcup [C_2] = [C_1 \sqcup C_2]$$

and generalise it to countably infinitely many commands/equivalence classes in the standard way.

The relation \approx_γ is an equivalence relation as it inherits reflexivity, symmetry and transitivity from the equality relation on sets, and \sqsubseteq_γ is a partial order on \mathbb{C}_γ as it inherits transitivity and reflexivity from set inclusion, while \approx_γ ensures anti-symmetry.

Furthermore, note that, by design of the language, we do not have to bring the outcome o into the equivalence relation, as faulting states can be distinguished from successful ones by having the dedicated program variable err in the store, and language errors can be distinguished from the missing resource errors by the value that err holds.

LEMMA D.10 (DOMAIN PROPERTY). *For any function implementation context γ , $(\mathbb{C}_\gamma, \sqsubseteq_\gamma)$ is a domain.*

PROOF. Since we have already argued the partial order property, there are only remaining two properties to show:

Chain-Closedness. For any chain $([C_n])_{n \in \mathbb{N}} \subseteq \mathbb{C}_\gamma$, its supremum is defined as $[\sqcup(C_n \mid n \in \mathbb{N})]$. Per definition of \mathbb{C} we have $\sqcup(C_n \mid n \in \mathbb{N}) \in \mathbb{C}$ which implies $[\sqcup(C_n \mid n \in \mathbb{N})] \in \mathbb{C}_\gamma$.

Least Element. The least element of \mathbb{C}_γ , denoted by \perp_γ , is the equivalence class of commands which, given the function implementation context γ , do not terminate on any state. One such representative is the command $C = \text{while } (\text{true}) \text{ skip}$. Since $\{(\sigma, \sigma') \in \text{State}^2 \mid \sigma, \text{while } (\text{true}) \text{ skip} \Downarrow_\gamma \sigma'\} = \emptyset$, we trivially obtain $\perp_\gamma \sqsubseteq_\gamma [C]$, for all $[C] \in \mathbb{C}_\gamma$. \square

LEMMA D.11 (SCOPE AND FUNCTION CALL EQUIVALENCE). *Given a function implementation context γ and a function f such that $\gamma(f) = (\vec{x}, C_f, E')$, it holds that*

$$\text{scope}((\vec{x}, \vec{E}), C_f, (\gamma, E')) \simeq_\gamma \gamma := f(\vec{E})$$

PROOF. We show in detail the case of successful execution; the faulting cases are analogous. Let $\gamma(f) = (\vec{x}, C_f, E')$ and $(s, h), (s', h') \in \text{State}$, such that

$$(s, h), \text{scope}((\vec{x}, \vec{E}), C_f, (\gamma, E')) \Downarrow_\gamma (s', h').$$

The operational semantics of scope implies:

- $s' = s[\gamma \rightarrow v']$
- $\llbracket \vec{E} \rrbracket_s = \vec{v}$
- $\llbracket E' \rrbracket_{s_q} = v'$
- $s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}]$
- $(s_p, h), C_f \Downarrow_\gamma (s_q, h')$

Due to the assumption $\gamma(f) = (\vec{x}, C_f, E')$, we fulfil all conditions in the antecedent of the operational semantics of the function call, and therefore obtain $(s, h), \gamma := f(\vec{E}) \Downarrow_\gamma (s', h')$.

Now, let $(s, h), (s', h') \in \text{State}$ such that $(s, h), \gamma := f(\vec{E}) \Downarrow_\gamma (s', h')$. The operational semantics of the function call implies:

- $s' = s[\gamma \rightarrow v']$
- $\llbracket \vec{E} \rrbracket_s = \vec{v}$
- $\llbracket E' \rrbracket_{s_q} = v'$
- $s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}]$
- $(s_p, h), C_f \Downarrow_\gamma (s_q, h')$
- $\gamma(f) = (\vec{x}, C_f, E')$

Therefore, we fulfil all conditions in the antecedent of the operational semantics of scope, and therefore obtain

$$(s, h), \text{scope}((\vec{x}, \vec{E}), C_f, (\gamma, E')) \Downarrow_\gamma (s', h'). \quad \square$$

In the following, let C_i denote the implementation of the function f_i , and C^i denote the i -th component of a vector C .

Definition D.12 (Function Call Substitution). Given a command $\bar{C} \in \text{Cmd}$, a vector of n commands $C = (C^1, \dots, C^n) \in \mathbb{C}^n$, a vector of n functions $F = (f_1, \dots, f_n)$ and a function implementation context γ , such that $F \subseteq \text{dom}(\gamma)$, we define a function call substitution $\bar{C}[C, \gamma, F]$ recursively on the structure of \bar{C} , with $\gamma(f_i) = (\vec{x}_i, -, E_i)$:

- $(\text{if } (B) C_1 \text{ else } C_2)[C, \gamma, F] := \text{if } (B) \{C_1[C, \gamma, F]\} \text{ else } \{C_2[C, \gamma, F]\}$
- $(\text{while } (B) \bar{C})[C, \gamma, F] := \text{while } (B) \{\bar{C}[C, \gamma, F]\}$
- $(C_1; C_2)[C, \gamma, F] := C_1[C, \gamma, F]; C_2[C, \gamma, F]$
- $(\gamma := g(\vec{E})) [C, \gamma, F] := \begin{cases} \text{scope}((\vec{x}_i, \vec{E}), C^i, (\gamma, E_i)) & \text{if } f_i = g \\ \gamma := g(\vec{E}) & \text{otherwise,} \end{cases}$
- $\bar{C}[C, \gamma, F] := \bar{C}$, for all other \bar{C} .

LEMMA D.13 (SUBSTITUTION PRESERVES \simeq_γ). *Given $I = \{1, \dots, n\}$, $F = (f_1, \dots, f_n)$, and γ such that $\forall i \in I. \gamma(f_i) = (-, C_i, -)$, $C = (C_1, \dots, C_n)$, and $i \in I$, it holds that*

$$C_i \simeq_\gamma C_i[C, \gamma, F]$$

PROOF. We prove the statement by structural induction on C_i . Per definition of function implementation contexts, $C_i \in \text{Cmd}$ and is therefore finite. The only non-trivial cases are the structural commands and the function call on f_i , as the substitution is the identity otherwise.

If-Else. $C_i = \text{if } (E) C_t \text{ else } C_f$. Let $\sigma, \sigma' \in \text{State}$ such that $\sigma, \text{if } (E) C_t \text{ else } C_f \Downarrow_Y \sigma'$, and let $\sigma = (s, h)$. Then, the operational semantics implies

$$\begin{aligned} & (\llbracket E \rrbracket_s = \text{true} \wedge (s, h), C_t \Downarrow_Y \sigma') \vee (\llbracket E \rrbracket_s = \text{false} \wedge (s, h), C_f \Downarrow_Y \sigma') \\ \stackrel{(\text{IH})}{\Leftrightarrow} & (\llbracket E \rrbracket_s = \text{true} \wedge (s, h), C_t[C, \gamma, F] \Downarrow_Y \sigma') \vee (\llbracket E \rrbracket_s = \text{false} \wedge (s, h), C_f[C, \gamma, F] \Downarrow_Y \sigma') \end{aligned}$$

which is equivalent to $(s, h), \text{if } (E) C_t[C, \gamma, F] \text{ else } C_f[C, \gamma, F] \Downarrow_Y \sigma'$. The faulting case is proven analogously to the successful case, and the while loop and the sequencing are proven analogously to if-else.

Function Call. We have to show that

$$(\gamma := f_i(\vec{E})) \simeq_Y \text{scope}(\vec{x}_i, \vec{E}), C_i, (\gamma, E_i)$$

where $\gamma(f_i) = (\vec{x}_i, C_i, E_i)$, but this holds directly due to Lemma D.11. \square

Before moving on to the Scott instantiation, we define a notion of (recursion) depth, which keeps track of the maximum number of nested function calls during the execution of commands.

Definition D.14 (Depth). Given a command $C \in \text{Cmd}$, a vector of functions $F = (f_1, \dots, f_n)$ and a derivation $\sigma, C \Downarrow_Y o : \sigma'$, we define $\text{depth}_F(\sigma, C \Downarrow_Y o : \sigma')$ inductively on the structure of big-step derivations of C as follows, noting that we extend the notion of a maximal element to the empty set by defining it to be zero:

$$\begin{aligned} & \text{depth}_F((s, h), \text{if } (E) C_t \text{ else } C_f \Downarrow_Y o : \sigma') \triangleq \\ \text{If-Else.} & \max \left(\begin{array}{l} \max\{\text{depth}_F((s, h), C_t \Downarrow_Y o : \sigma') \mid \llbracket E \rrbracket_s = \text{true}\}, \\ \max\{\text{depth}_F((s, h), C_f \Downarrow_Y o : \sigma') \mid \llbracket E \rrbracket_s = \text{false}\} \end{array} \right) \\ & \text{depth}_F(\sigma, C_1; C_2 \Downarrow_Y o : \sigma') \triangleq \\ \text{Sequence.} & \max \left(\begin{array}{l} \max\{\text{depth}_F(\sigma, C_1 \Downarrow_Y \bar{\sigma}), \text{depth}_F(\bar{\sigma}, C_2 \Downarrow_Y o : \sigma') \mid \\ \sigma, C_1 \Downarrow_Y \bar{\sigma} \wedge \bar{\sigma}, C_2 \Downarrow_Y o : \sigma'\}, \\ \max\{\text{depth}_F(\sigma, C_1 \Downarrow_Y o : \sigma') \mid o = \text{err/miss} \wedge \sigma, C_1 \Downarrow_Y o : \sigma'\} \end{array} \right) \\ & \text{depth}_F((s, h), \text{while } (E) C \Downarrow_Y o : \sigma') \triangleq \\ \text{While.} & \max \left(\begin{array}{l} \max\{\text{depth}_F(\sigma, C \Downarrow_Y \bar{\sigma}), \text{depth}_F(\bar{\sigma}, \text{while } (E) C \Downarrow_Y o : \sigma') \mid \\ \llbracket E \rrbracket_s = \text{true} \wedge \sigma, C \Downarrow_Y \bar{\sigma} \wedge \bar{\sigma}, \text{while } (E) C \Downarrow_Y o : \sigma'\}, \\ \max\{\text{depth}_F(\sigma, C \Downarrow_Y o : \sigma') \mid o = \text{err/miss} \wedge \llbracket E \rrbracket_s = \text{true} \wedge \sigma, C \Downarrow_Y o : \sigma'\} \end{array} \right) \end{aligned}$$

Function Call. $\text{depth}_F((s, h), \gamma := g(\vec{E}), \Downarrow_Y o : (s', h')) \triangleq 0, \quad \text{if } \forall i \in I. g \neq f_i$

$$\begin{aligned} & \text{depth}_F((s, h), \gamma := f_i(\vec{E}), \Downarrow_Y o : (s', h')) \triangleq \\ & \max \left(\begin{array}{l} \max\{1 + \text{depth}_F((s_p, h), C_i \Downarrow_Y (s_q, h')) \mid \\ o = \text{ok}, \llbracket \vec{E} \rrbracket_s = \vec{v}, (s_p, h), C_f \Downarrow_Y (s_q, h'), \llbracket E_i \rrbracket_{s_q} = v'\}, \\ \max\{1 + \text{depth}_F((s_p, h), C_i \Downarrow_Y o : (s_q, h')) \mid \\ o = \text{err/miss}, \llbracket \vec{E} \rrbracket_s = \vec{v}, (s_p, h), C_f \Downarrow_Y o : (s_q, h')\}, \\ \max\{1 + \text{depth}_F((s_p, h), C_i \Downarrow_Y o : (s_q, h')) \mid \\ o = \text{err}, \llbracket \vec{E} \rrbracket_s = \vec{v}, (s_p, h), C_f \Downarrow_Y (s_q, h'), \llbracket E_i \rrbracket_{s_q} = \perp\}, \end{array} \right) \end{aligned}$$

where $\gamma(f_i) = (\vec{x}_i, C_i, E_i)$, and s_p and s' are defined as in the operational semantics of the function call.

Remaining Commands. $\text{depth}_F(\sigma, C \Downarrow_Y o : \sigma') \triangleq 0$.

D.1 1-dimensional Scott Instantiation

The env-extend rule allows us to add a set of n functions to a given valid environment. Soundness of this rule is proven in Appendix E through transfinite induction. In each of the cases (zero, successor ordinal, limit ordinal), a Scott induction is required. In Appendix D.2, we present and prove the Scott induction required to show soundness of the env-extend rule. Here, we present the Scott induction as required for the case where only *one* function is added to a given environment at a time.

The general proof, as presented in Appendix D.2 evolves naturally from and relies heavily on this simpler case, while introducing heavier notation. To minimize clutter in later definitions, we introduce the over-approximation quadruple $\{P\} C \{ok : Q_{ok}\} \{err : Q_{err}\}$ and define its notion of validity.

Definition D.15 (OX-Validity). Given an OX-quadruple $\{P\} C \{ok : Q_{ok}\} \{err : Q_{err}\}$ and a function we define for an arbitrary implementation context γ

$$\begin{aligned} \gamma \models \{P\} C \{ok : Q_{ok}\} \{err : Q_{err}\} &\iff \\ &\forall \theta, s, h, o, s', h'', h_f. \theta, s, h \models P \\ &\implies (s, h \uplus h_f), C \Downarrow_{\gamma} o : (s', h'') \\ &\implies o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models Q_o \end{aligned}$$

and for an arbitrary specification context Γ

$$\begin{aligned} \Gamma \models \{P\} C \{ok : Q_{ok}\} \{err : Q_{err}\} &\iff \\ \forall \gamma. \models (\gamma, \Gamma) \implies \gamma \models \{P\} C \{ok : Q_{ok}\} \{err : Q_{err}\} \end{aligned}$$

In the following, we will also write $\{P\} C \{Q\}$ as a shorthand for $\{P\} C \{ok : Q_{ok}\} \{err : Q_{err}\}$. Onward, we assume the following:

- (A1) a valid environment, $\models (\gamma, \Gamma)$;
- (A2) a function $f(\vec{x})\{C_f; \text{return } E'\}$ that is not in the domain of γ ;
- (A3) an arbitrary element $\alpha \in \mathcal{O}$;
- (A4) a set of terminating (external) specifications for f , $\{(P(\beta)) (Q(\beta)) \mid \beta < \alpha\}$;
- (A5) a set of non-terminating (external) specifications for f , $\{(P_{\infty}(\beta)) (\text{False}) \mid \beta \leq \alpha\}$;
- (A6) an extension of γ with f : $\gamma' \triangleq \gamma[f \mapsto (\vec{x}, C_f, E')]$; and
- (A7) an extension of Γ with the given specifications of f :

$$\Gamma(\alpha) \triangleq \Gamma[f \mapsto \{(P(\beta)) (Q(\beta)) \mid \beta < \alpha\} \cup \{(P_{\infty}(\beta)) (\text{False}) \mid \beta \leq \alpha\}]$$

We next define the function $g : \mathbb{C}_{\gamma'} \rightarrow \mathbb{C}_{\gamma'}$, to be used in the upcoming Scott induction, as follows:

$$g([C]) \stackrel{\text{def}}{=} [h(C)]$$

where $h : \mathbb{C} \rightarrow \mathbb{C}$ is defined as $h(C) := C_f[C, \gamma', f]$.

Intuitively, h takes an arbitrary command C from \mathbb{C} as an argument and substitutes it for any function call on f in function body C_f . The function g then lifts this operation to the quotient space $\mathbb{C}_{\gamma'}$. The definitions of h and g trivially yield the following identities for arbitrary $C \in \mathbb{C}$ and $(C_n)_{n \in \mathbb{N}} \subseteq \mathbb{C}$:

$$\begin{aligned} \text{(G1)} \quad \bigsqcup_{n \in \mathbb{N}} g([C_n]) &= [\bigsqcup_{n \in \mathbb{N}} h(C_n)] \\ \text{(G2)} \quad \bigsqcup_{n \in \mathbb{N}} g^n([C]) &= [\bigsqcup_{n \in \mathbb{N}} h^n(C)] \end{aligned}$$

LEMMA D.16. *The function g is continuous.*

PROOF. We begin by proving monotonicity.

Monotonicity. We prove the monotonicity of h . We need to show that for all $C_1, C_2 \in \mathbb{C}$, it holds that

$$C_1 \sqsubseteq_{\gamma'} C_2 \implies C_f[C_1, \gamma', f] \sqsubseteq_{\gamma'} C_f[C_2, \gamma', f]$$

Let $\sigma, \sigma' \in \text{State}$ such that $\sigma, C_f[C_1, \gamma', f] \Downarrow_{\gamma'} o : \sigma'$. We need to show that

$$\sigma, C_f[C_2, \gamma', f] \Downarrow_{\gamma'} o : \sigma'$$

and we do so by structural induction on C_f . The only non-trivial cases are the compound commands and the function call, as the substitution is the identity for all other cases.

If-Else. $C_f = \text{if } (E) C_t \text{ else } C_f$. Let $(s, h), \sigma' \in \text{State}$ such that

$$(s, h), \text{if } (E) C_t[C_1, \gamma', f] \text{ else } C_f[C_1, \gamma', f] \Downarrow_{\gamma'} o : \sigma'$$

The operational semantics yields (for an appropriate σ_{err}):

$$\begin{aligned} & (\llbracket E \rrbracket_s = \text{true} \wedge (s, h), C_t[C_1, \gamma', f] \Downarrow_{\gamma'} o : \sigma') \vee \\ & (\llbracket E \rrbracket_s = \text{false} \wedge (s, h), C_f[C_1, \gamma', f] \Downarrow_{\gamma'} o : \sigma') \vee \\ & (\llbracket E \rrbracket_s = \perp \wedge \sigma' = \sigma_{err}) \\ \Rightarrow & \text{(IH)} \quad (\llbracket E \rrbracket_s = \text{true} \wedge (s, h), C_t[C_2, \gamma', f] \Downarrow_{\gamma'} o : \sigma') \vee \\ & (\llbracket E \rrbracket_s = \text{false} \wedge (s, h), C_f[C_2, \gamma', f] \Downarrow_{\gamma'} o : \sigma') \vee \\ & (\llbracket E \rrbracket_s = \perp \wedge \sigma' = \sigma_{err}) \end{aligned}$$

which implies the desired

$$(s, h), \text{if } (E) C_t[C_2, \gamma', f] \text{ else } C_f[C_2, \gamma', f] \Downarrow_{\gamma'} o : \sigma'$$

The while loop and the sequencing cases are proven analogously.

Function Call. $C_f = \gamma := f(\vec{E})$. Using Lemma D.11 and considering the successful case only as the faulting cases are proven analogously, let $\sigma, \sigma' \in \text{State}$ such that

$$\sigma, \text{scope}((x, \vec{E}), C_1, (\gamma, E')) \Downarrow_{\gamma'} \sigma'$$

Letting $\sigma = (s, h)$ and $\sigma' = (s', h')$, the operational semantics for scope then implies

$$\begin{aligned} & s' = s[\gamma \rightarrow v'] \wedge \llbracket \vec{E} \rrbracket_s = \vec{v} \wedge \llbracket E' \rrbracket_{s_q} = v' \wedge \text{pv}(C) \setminus \{\vec{x}\} = \vec{z} \\ & \wedge s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \wedge (s_p, h), C_1 \Downarrow_{\gamma'} (s_q, h') \\ \xRightarrow{C_1 \sqsubseteq_{\gamma'} C_2} & s' = s[\gamma \rightarrow v'] \wedge \llbracket \vec{E} \rrbracket_s = \vec{v} \wedge \llbracket E' \rrbracket_{s_q} = v' \wedge \text{pv}(C) \setminus \{\vec{x}\} = \vec{z} \\ & \wedge s_p = \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}] \wedge (s_p, h), C_2 \Downarrow_{\gamma'} (s_q, h') \end{aligned}$$

which implies the desired

$$\sigma, \text{scope}((x, \vec{E}), C_2, (\gamma, E')) \Downarrow_{\gamma'} \sigma'$$

The monotonicity of g follows straightforwardly from the monotonicity of h .

Supremum-Preservation. Assume a chain $(C_n)_{n \in \mathbb{N}}$ in \mathbb{C} . First, we show that $\bigsqcup_{n \in \mathbb{N}} h(C_n) \sqsubseteq_{\gamma'} h(\bigsqcup_{n \in \mathbb{N}} C_n)$:

$$\begin{aligned} & \sigma, \bigsqcup_{n \in \mathbb{N}} h(C_n) \Downarrow_{\gamma'} o : \sigma' \\ \Rightarrow & \exists m \in \mathbb{N}. \sigma, h(C_m) \Downarrow_{\gamma'} o : \sigma' \\ \Rightarrow & \exists m \in \mathbb{N}. \sigma, C_f[C_m, \gamma', f] \Downarrow_{\gamma'} o : \sigma' \\ \Rightarrow & \sigma, C_f[\bigsqcup_{n \in \mathbb{N}} C_n, \gamma', f] \Downarrow_{\gamma'} o : \sigma' \\ \Rightarrow & \sigma, h(\bigsqcup_{n \in \mathbb{N}} C_n) \Downarrow_{\gamma'} o : \sigma' \end{aligned}$$

Next, we show that $h(\bigsqcup_{n \in \mathbb{N}} C_n) \sqsubseteq_{\gamma'} \bigsqcup_{n \in \mathbb{N}} h(C_n)$. Let $\sigma, h(\bigsqcup_{n \in \mathbb{N}} C_n) \Downarrow_{\gamma'} \sigma'$, i.e. $\sigma, C_f[\bigsqcup_{n \in \mathbb{N}} C_n, \gamma', f] \Downarrow_{\gamma'} \sigma'$.

Then, since $C_f \in \text{Cmd}$, it is a finite command and hence has a finite number t of function calls on f . At each function call substitution site, the execution will execute some command C_n . Assume $k_1, \dots, k_t \in \mathbb{N}$ such that at the i -th execution site, the command C_{k_i} is executed, and let $k \triangleq \max(k_1, \dots, k_t)$. Since $(C_n)_{n \in \mathbb{N}}$ is a chain, we have that $C_{k_i} \sqsubseteq_{\gamma'} C_k$ for all $i \in \{1, \dots, t\}$ and therefore:

$$\begin{aligned} & \sigma, C_f[C_k, \gamma', f] \Downarrow_{\gamma'} \sigma' \\ \Rightarrow & \sigma, \bigsqcup_{n \in \mathbb{N}} C_f[C_n, \gamma', f] \Downarrow_{\gamma'} \sigma' \\ \Rightarrow & \sigma, \bigsqcup_{n \in \mathbb{N}} h(C_n) \Downarrow_{\gamma'} o : \sigma' \end{aligned}$$

The supremum preservation of g follows trivially from the supremum preservation of h . \square

Next, we introduce the admissible set we will use in this instantiation of the Scott induction.

LEMMA D.17 (ADMISSIBLE SUBSET S^α). *The set S^α , defined as*

$$S^\alpha := \{[\tilde{C}] \in \mathbb{C}_{\gamma'} \mid \exists C \in [\tilde{C}]. \forall t \in (\Gamma(\alpha))(f). \exists (P') (Q') \in \text{Int}_{\gamma',f}(t). \Gamma \models \{P'\} C \{Q'\}\}$$

is an admissible subset of $(\mathbb{C}_{\gamma'}, \sqsubseteq_{\gamma'})$.

PROOF. Least Element. We know that $\perp_{\gamma'} = [\text{while (true) skip}]$ and that this commands trivially semantically satisfies any OX-quadruple. Therefore, $\perp_{\gamma'} \in S^\alpha$.

Chain-Closure. We need to show that given an arbitrary chain $([C'_n])_{n \in \mathbb{N}} \subseteq S^\alpha$, it holds that $\sqcup([C'_n] \mid n \in \mathbb{N}) \in S^\alpha$. Onwards, we will use the following notation:

- $(P) (Q) \triangleq (P) (ok : Q_{ok})(err : Q_{err})$
- $(P_n) (Q_n) \triangleq (P_n) (ok : Q_{ok}^n)(err : Q_{err}^n)$

The definition of S^α yields the existence of a chain $(C_n)_{n \in \mathbb{N}} \subseteq \mathbb{C}$ such that for all $n \in \mathbb{N}$, it holds that $C_n \in [C'_n]$ and

$$\forall n \in \mathbb{N}, (P) (Q) \in ((\Gamma(\alpha))(f). \exists (P_n) (Q_n) \in \text{Int}_{\gamma',f}((P) (Q)). \Gamma \models \{P_n\} C_n \{Q_n\})$$

Per definition of Int , we know that $P_n = P \star \bar{z} = \text{null}$ for all $n \in \mathbb{N}$. Together with the definition of choice, we obtain that

$$\Gamma \models \{P \star \bar{z} = \text{null}\} \sqcup (C_n \mid n \in \mathbb{N}) \{ok : \bigvee_{n \in \mathbb{N}} Q_{ok}^n\} \{err : \bigvee_{n \in \mathbb{N}} Q_{err}^n\}$$

It remains to show that $(P \star \bar{z} = \text{null}) (\bigvee_{n \in \mathbb{N}} Q_n)$ is an internalisation of $(P) (Q)$. Since $(P \star \bar{z} = \text{null}) (Q_n)$ are internalisations of $(P) (Q)$, we obtain

$$\forall n \in \mathbb{N}. (Q_{ok} \Leftrightarrow \exists \vec{p}. Q_{ok}^n[\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}]) \wedge (Q_{err} \Leftrightarrow \exists \vec{p}. Q_{err}^n[\vec{p}/\vec{p}])$$

This implies

$$\begin{aligned} Q_{ok} &\Leftrightarrow \bigvee_{n \in \mathbb{N}} (\exists \vec{p}. Q_{ok}^n[\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}]) \\ &\Leftrightarrow \exists \vec{p}. \bigvee_{n \in \mathbb{N}} (Q_{ok}^n[\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}]) \\ &\Leftrightarrow \exists \vec{p}. (\bigvee_{n \in \mathbb{N}} Q_{ok}^n[\vec{p}/\vec{p}]) \star \text{ret} = E'[\vec{p}/\vec{p}] \\ &\Leftrightarrow \exists \vec{p}. (\bigvee_{n \in \mathbb{N}} Q_{ok}^n) [\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}] \end{aligned}$$

and analogously

$$\begin{aligned} Q_{err} &\Leftrightarrow \bigvee_{n \in \mathbb{N}} (\exists \vec{p}. Q_{err}^n[\vec{p}/\vec{p}]) \\ &\Leftrightarrow \exists \vec{p}. \bigvee_{n \in \mathbb{N}} (Q_{err}^n[\vec{p}/\vec{p}]) \\ &\Leftrightarrow \exists \vec{p}. (\bigvee_{n \in \mathbb{N}} Q_{err}^n) [\vec{p}/\vec{p}] \end{aligned}$$

Therefore, $\sqcup(C_n \mid n \in \mathbb{N}) \in S^\alpha$, which yields $\sqcup([C_n] \mid n \in \mathbb{N}) \in S^\alpha$ and finally $\sqcup([C'_n] \mid n \in \mathbb{N}) \in S^\alpha$. \square

This concludes the set-up for Scott induction, which allows us to prove the inductive step.

LEMMA D.18 (SCOTT CONDITION). *Under the assumptions (A1)-(A7) and additionally assuming*

$$\forall t \in (\Gamma(\alpha))(f). \exists t' \in \text{Int}_{\gamma',f}(t). \Gamma(\alpha) \vdash C_f : t' \quad (1)$$

and

$$\exists t' \in \text{Int}_{\gamma',f}((P(\alpha)) (Q(\alpha))). \Gamma(\alpha) \vdash C_f : t' \quad (2)$$

it holds that $g(S^\alpha) \subseteq S^\alpha$.

PROOF. Let $[C] \in S^\alpha$. Therefore, there exists a $C' \in [C]$ such that for all $t \in (\Gamma(\alpha))(f)$ exists a $(P) (Q) \in \text{Int}_{\gamma',f}(t)$ such that **(H)** $\Gamma \models \{P\} C' \{Q\}$. This implies that C' does not call on f , because Γ holds no specifications for f . Hence, $C_f[C', \gamma', f]$ does not call on f either. We prove the statement by showing the more general claim

$$\Gamma(\alpha) \vdash (P) C_f (Q) \implies \Gamma \models \{P\} C_f[C', \gamma', f] \{Q\}$$

for arbitrary precondition P and postcondition Q by induction over the structure of C_f .

Base Commands and Function Calls on $g \neq f$. In this case, $C_f[C', \gamma', f] = C_f$ and C_f does not call on f . Therefore, the proof tree of $\Gamma(\alpha) \vdash (P) C_f (Q)$ uses no specifications on f , which implies that $\Gamma \vdash (P) C_f (Q)$. From (A1) $\vdash (\gamma, \Gamma)$, we obtain $\Gamma \vdash (P) C_f (Q)$, which implies $\Gamma \models \{P\} C_f \{Q\}$.

Compound Command Rules: if-then, if-else, sequence. If-then, if-else and sequence are proven directly, using the IH.

Compound Command Rules: while-iterate. Given the while rule

$$\frac{\text{WHILE-ITERATE} \quad \begin{array}{l} \forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B} \quad P_\infty \triangleq \text{False} \\ \forall i \in \mathbb{N}. \Gamma \vdash (P_i \wedge E) C (ok : P_{i+1}) (err : Q_i) \\ m \triangleq \min(\{i \in \mathbb{N} \cup \{\infty\} \mid \models P_i \Rightarrow \neg E\}) \end{array}}{\Gamma \vdash (P_0) \text{ while } (E) C (ok : P_m) (err : \exists n < m. Q_n)}$$

we assume

- (W1) $\theta, s, h \models P_0$
- (W2) $(s, h \uplus h_f), \text{while } (E) C[C', \gamma', f] \Downarrow_{\bar{\gamma}} (s', h'')$
- (W3) $\forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B}$
- (W4) $P_\infty \triangleq \text{False}$
- (W5) $\forall i \in \mathbb{N}. \Gamma(\alpha) \vdash (P_i \wedge E) C (ok : P_{i+1}) (err : Q_i)$
- (W6) $m \triangleq \min(\{i \in \mathbb{N} \cup \{\infty\} \mid \models P_i \Rightarrow \neg E\})$

Noting that $(\text{while } (E) C)[C', \gamma', f] = \text{while } (E) C[C', \gamma', f]$, (W2) implies that

- (W7) $(\llbracket E \rrbracket_{s, h \uplus h_f} = \text{false} \wedge (s, h \uplus h_f) = (s', h'')) \vee (\llbracket E \rrbracket_{s, h \uplus h_f} = \text{true} \wedge (s, h \uplus h_f), C[C', \gamma', f] \Downarrow_{\gamma'} \bar{\sigma} \wedge \bar{\sigma}, \text{while } (E) C[C', \gamma', f] \Downarrow_{\gamma'} (s', h''))$

and (W5) and the inductive hypothesis imply

$$(W8) \quad \forall i \in \mathbb{N}. \Gamma \models \{P_i \wedge E\} C[C', \gamma', f] \{ok : P_{i+1}\} \{err : Q_i\}$$

We know that $m \neq \infty$, as otherwise the loop would be non-terminating, contradicting (W2). If $m = 0$, then $\llbracket E \rrbracket_{\theta, s} = \text{false}$, and (W7) trivially yields the desired result.

Otherwise, we have that $m > 0$. Then, (W8) yields

$$\Gamma \models \{P_{i-1} \wedge E\} C[C', \gamma', f] \{ok : P_i\} \{err : Q_{i-1}\}$$

for all $0 < i \leq m$ and (W6) yields $\Gamma \models \{P_m\} \text{ while } (E) C[C', \gamma', f] \{P_m\}$. From here, applying the operational semantics of while m times, similarly to the proof in RHL [13], we obtain the desired

$$\Gamma \models \{P_0\} \text{ while } (E) C[C', \gamma', f] \{ok : P_m\} \{err : \exists n < m. Q_n\}$$

Structural rules. All four structural rules (equiv, exists, frame, and disj) are proven trivially, using the inductive hypothesis. We give the proof for the equivalence rule:

$$\frac{\text{EQUIV} \quad \begin{array}{l} \models P \Leftrightarrow P' \quad \Gamma(\alpha) \vdash (P') C_f (ok : Q'_{ok}) (err : Q'_{err}) \quad \models Q'_{ok} \Leftrightarrow Q_{ok} \quad \models Q'_{err} \Leftrightarrow Q_{err} \end{array}}{\Gamma(\alpha) \vdash (P) C_f (ok : Q_{ok}) (err : Q_{err})}$$

where the IH gives us $\Gamma \models \{P'\} C_f[C', \gamma', f] \{ok : Q'_{ok}\} \{err : Q'_{err}\}$, from which the desired claim is obtained trivially.

Function call on f . ($\gamma := f(\vec{E})$) $[C', \gamma', f] = \text{scope}((\vec{x}, \vec{E}), C', (\gamma, E'))$, where $\gamma'(f) = (\vec{x}, C_f, E')$. Therefore, we need to show that

$$\Gamma \models \{P\} \text{ scope}((\vec{x}, \vec{E}), C', (\gamma, E')) \{Q\}$$

The assumption $\Gamma(\alpha) \vdash (P) \gamma := f(\vec{E}) (Q)$ implies via the fcall rule that $P = (\gamma = E_y \star \vec{E} = \vec{x} \star P^*)$, where P^* is the program-variable-free part either the pre-condition P_∞ of the non-terminating or the pre-condition P of the (partially) terminating specification, and that $(\vec{x} = \vec{x} \star P^*) (Q) \in (\Gamma(\alpha))(f)$. We assume the following:

- (F0) an arbitrary $\bar{\gamma}$ such that $\models (\bar{\gamma}, \Gamma(\alpha))$
- (F1) $\theta, s, h \models \gamma = E_y \star \vec{E} = \vec{x} \star P^*$
- (F2) arbitrary h_f and h'' such that $(s, h \uplus h_f), \text{scope}((\vec{x}, \vec{E}), C', (\gamma, E')) \Downarrow_{\bar{\gamma}} o : (s', h'')$

We need to show that

$$(o \neq \text{miss}) \wedge (\exists h'. h'' = h' \uplus h_f \wedge ((o = \text{ok} \wedge \theta, s', h' \models Q_{\text{ok}}) \vee (o = \text{err} \wedge \theta, s', h' \models Q_{\text{err}})))$$

Defining

$$\text{(F3)} \quad \vec{v} := \llbracket \vec{E} \rrbracket_s$$

$$\text{(F4)} \quad \vec{z} := \text{pv}(C') \setminus \{\vec{x}\}.$$

$$\text{(F5)} \quad s_p := \emptyset[\vec{x} \rightarrow \vec{v}][\vec{z} \rightarrow \text{null}],$$

the operational semantics of scope and (F2) imply

$$\text{(F6)} \quad (s_p, h \uplus h_f, C' \Downarrow_{\vec{y}} o : (s_q, h''))$$

$$\text{(F7a)} \quad o = \text{ok} \Rightarrow (s' = s[\gamma \rightarrow \llbracket E' \rrbracket_{s_q}])$$

$$\text{(F7b)} \quad o = \text{err} \Rightarrow (s' = s[\text{err} \rightarrow \llbracket \text{err} \rrbracket_{s_q}])$$

The definition of S^α implies existence of a (P') $(ok : Q'_{\text{ok}})(err : Q'_{\text{err}}) \in \text{Int}_{\gamma, f}((P^*)(Q))$ such that

$$\text{(H8a)} \quad \Gamma \models \{P'\} C' \{Q'\}. \text{ (A1) and (F0) imply } \models (\vec{y}, \Gamma), \text{ which yields with (H8a) that } \text{(H8b)} \quad \vec{y} \models \{P'\} C' \{Q'\}$$

Per definition of the internalisation, we know that $P' = \vec{x} = \vec{x} \star P^* \star \vec{z} = \text{null}$, which implies with (F3)-(F5) that **(H9)** $\theta, s_p, h \models P'$. Then, (F6), (F8b) and (F9) imply

$$\text{(H10)} \quad (o \neq \text{miss}) \wedge (\exists h'. h'' = h' \uplus h_f \wedge ((o = \text{ok} \wedge \theta, s_q, h' \models Q'_{\text{ok}}) \vee (o = \text{err} \wedge \theta, s_q, h' \models Q'_{\text{err}})))$$

This yields $\theta, s', h' \models Q_o$ as Q'_{ok} and Q'_{err} are internalisations of Q_{ok} and Q_{err} respectively, which in turn implies the desired result, i.e.

$$\Gamma \models \{P\} \text{scope}((\vec{x}, \vec{E}), C', (\gamma, E')) \{Q\}$$

This concludes the proof of the general statement. Instantiating it with the existentially quantified t' from (1) and (2) then yields the desired result

$$[C] \in S^\alpha \implies [h(C)] \in S^\alpha$$

i.e., the Scott condition $g(S^\alpha) \subseteq S^\alpha$.

□

Finally, we need to show that the function body C_f of f is indeed equivalent to the least fixpoint of g , denoted by $\text{lfp}(g)$.

LEMMA D.19 (SCOTT'S LAST STEP). *The function body C_f is in the least fixpoint of g , i.e.*

$$C_f \in \text{lfp}(g)$$

Again, we prove a slightly different statement first, and then apply it to prove the lemma. Onward, we write $\perp_{\gamma'}$ to denote the least element of $\mathbb{C}_{\gamma'}$ (and also of S^α). Onwards, we will use wts as a shorthand for the command while (true) skip. Keep in mind that $\perp_{\gamma'} = [\text{wts}]$.

LEMMA D.20. *For all $n \in \mathbb{N}$, it holds that*

$$\forall \sigma, \sigma' \in \text{State}. \sigma, C_f \Downarrow_{\gamma'} \sigma' \wedge \text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') \leq n \iff \sigma, h^{n+1}(\text{wts}) \Downarrow_{\gamma'} \sigma'$$

where do not explicitly include the outcome statement o to avoid clutter.

PROOF. By induction on n .

Base Case: $n = 0$.

" \Rightarrow ": Let $\sigma, C_f \Downarrow_{\gamma'} \sigma' \wedge \text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') = 0$. Since $C_f \in \text{Cmd}$, it cannot include the scope command. Since the depth is zero, the execution path does not reach a function call on f . Hence, this call may be replaced by any other command, including wts , i.e.

$$\begin{aligned} & \sigma, C_f[\text{wts}, \gamma', f] \Downarrow_{\gamma'} \sigma' \\ \Leftrightarrow & \sigma, h(\text{wts}) \Downarrow_{\gamma'} \sigma' \end{aligned}$$

" \Leftarrow ": Assuming $\sigma, h(\text{wts}) \Downarrow_{\gamma'} \sigma'$, we obtain $\sigma, C_f[\text{wts}, \gamma', f] \Downarrow_{\gamma'} \sigma'$. Since wts does not terminate on any state, this implies that the execution does not reach the command wts , that is, it does not reach any function call site of f and, therefore, we obtain $\sigma, C_f \Downarrow_{\gamma'} \sigma'$ and $\text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') = 0$.

Inductive Step. Assume that the equivalence holds for some $n \in \mathbb{N}$, and the goal is then to prove that it holds for $n + 1$.

" \Rightarrow ": Let $\sigma, C_f \Downarrow_{\gamma'} \sigma' \wedge \text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') \leq n + 1$. If $\text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') < n + 1$, the inductive hypothesis yields $\sigma, h^{n+1}(\text{wts}) \Downarrow_{\gamma'} \sigma'$. Since $h^{n+1}(\text{wts})$ implies that the execution terminates and therefore does not reach any instance of the command wts , we may substitute wts for any other command, including $h(\text{wts})$. This yields $\sigma, h^{n+2}(\text{wts}) \Downarrow_{\gamma'} \sigma'$.

Finally, let $\text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') = n + 1$ and let $\gamma := f(\vec{x})$ be an arbitrary function call on f reached by the execution. That means that there exist states σ_1 and σ_2 , such that the execution of the initial part of C_f from σ up to that function call yields σ_1 , that $\sigma_1, \gamma := f(\vec{x}) \Downarrow_{\gamma'} \sigma_2$, and that the execution of the remaining part of C_f on σ_2 yields σ' . By instantiating the operational semantics of the function call with σ_1 and σ_2 , we obtain $(s_p, h), C_f \Downarrow_{\gamma'} (s_q, h')$, where, per definition of the depth function, $\text{depth}_F((s_p, h), C_f \Downarrow_{\gamma'} (s_q, h')) = n$. The inductive hypothesis then implies $(s_p, h), h^{n+1}(\text{wts}) \Downarrow_{\gamma'} (s_q, h')$. Therefore, $\sigma, C_f[h^{n+1}(\text{wts}), \gamma', f] \Downarrow_{\gamma'} \sigma'$ and since this command does not call on f and given the definition of h , we obtain $\sigma, h^{n+2}(\text{wts}) \Downarrow_{\gamma'} \sigma'$.

" \Leftarrow ": Assume $\sigma, h^{n+2}(\text{wts}) \Downarrow_{\gamma'} \sigma'$: that is, $\sigma, C_f[h^{n+1}(\text{wts}), \gamma', f] \Downarrow_{\gamma'} \sigma'$. If there are no substitution sites in C_f , the desired goal is obtained trivially, as the substitution is vacuous and the considered depth is zero by definition. Otherwise, consider an arbitrary substitution site in C_f reached by the execution starting from σ , and let the state before the substitution site be some σ_1 . By instantiating the operational semantics of scope, we obtain s_p, s_q, h' and σ_2 , such that $(s_p, h), h^{n+1}(\text{wts}) \Downarrow_{\gamma'} (s_q, h')$ and the remaining part of $C_f[h^{n+1}(\text{wts}), \gamma', f]$ executed on σ_2 terminates in σ' . Per the inductive hypothesis, we have that $(s_p, h), C_f \Downarrow_{\gamma'} (s_q, h')$ and $\text{depth}_F((s_p, h), C_f \Downarrow_{\gamma'} (s_q, h')) \leq n$. This implies, given the operational semantics of function call and the definition of depth, that $\sigma, C_f \Downarrow_{\gamma'} (s_q, h')$ and $\text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') \leq n + 1$, which concludes the proof. \square

With this in place, we can prove Lemma D.19, concluding the overall proof:

PROOF OF LEMMA D.19. We know that the least fixpoint of g obeys the following identity:

$$\text{lfp}(g) = \bigsqcup_{n \in \mathbb{N}} g^n(\perp_{\gamma'}) = [\bigsqcup_{n \in \mathbb{N}} h^n(\text{wts})] \quad (3)$$

To prove the statement of the lemma, we will show that

$$\bigsqcup_{n \in \mathbb{N}} h^n(\text{wts}) \simeq_{\gamma'} C_f$$

" \Rightarrow ": Assume $\sigma, \sigma' \in \text{State}$ such that $\sigma, \bigsqcup_{n \in \mathbb{N}} h^n(\text{wts}) \Downarrow_{\gamma'} \sigma'$. Therefore, there exists an $n \in \mathbb{N}$ such that $\sigma, h^n(\text{wts}) \Downarrow_{\gamma'} \sigma'$. Since the command wts does not terminate on any state and since $h^0(\text{wts}) = \text{wts}$, n must be strictly positive. Lemma D.20 then implies that $\sigma, C_f \Downarrow_{\gamma'} \sigma'$, which concludes the proof.

" \Leftarrow ": Assuming $\sigma, \sigma' \in \text{State}$ such that $\sigma, C_f \Downarrow_{\gamma'} \sigma'$, we know that C_f terminates when executed on σ , and therefore has a finite execution depth: that is, $\text{depth}_F(\sigma, C_f \Downarrow_{\gamma'} \sigma') = n$ holds for some $n \in \mathbb{N}$. Lemma D.20 then implies $\sigma, h^{n+1}(\text{wts}) \Downarrow_{\gamma'} \sigma'$, and therefore $\sigma, \bigsqcup_{n \in \mathbb{N}} h^n(\text{wts}) \Downarrow_{\gamma'} \sigma'$.

This yields $C_f \in \text{lfp}(g)$. \square

Theorem D.5 and Lemmas D.18 and D.19 finally finimplly that $[C_f] \in S^\alpha$.

D.2 n-dimensional Scott Instantiation

The instantiation presented so far suffices to prove soundness for recursive functions which potentially have both terminating and non-terminating specifications. However, we wish to allow clusters of *mutually* recursive functions, and hence require a more general instantiation of the Scott induction. In particular, given an environment (γ, Γ) , we add on a mutually recursive cluster $F := (f_1, \dots, f_n)$ of n functions.

We assume the following:

(B1) a valid environment, $\models (\gamma, \Gamma)$;

(B2) a set of n functions $f_i(\vec{x}_i)\{C_i; \text{return } E_i\}$ with $i \in I \triangleq \{1, \dots, n\}$ that is not in the domain of γ ;

- (B3) an arbitrary element $\alpha \in \mathcal{O}$;
- (B4) a set of terminating (external) specifications for each f_i , $\{(P^i(\beta)) \ (Q^i(\beta)) \mid \beta < \alpha\}$;
- (B5) a set of non-terminating (external) specifications for each f_i , $\{(P_\infty^i(\beta)) \ (\text{False}) \mid \beta \leq \alpha\}$;
- (B6) an extension of $\gamma: \gamma' \triangleq \gamma[f_i \mapsto (\tilde{x}_i, C_i, E_i)]_{i \in I}$; and
- (B7) an extension of Γ with the given specifications:

$$\Gamma(\alpha) \triangleq \Gamma[f_i \mapsto \{(P^i(\beta)) \ (Q^i(\beta)) \mid \beta < \alpha\} \cup \{(P_\infty^i(\beta)) \ (\text{False}) \mid \beta \leq \alpha\}]_{i \in I}$$

Note on Notation. As most elements we will be using in this section are n -tuples of commands, or chains of such n -tuples, we use the following notation:

- C_i : a subscript i denotes that the commands C_i is the implementation of the function f_i , as recorded in the associated function implementation context γ' ,
- C^i : a superscript i denotes the i -th component of an n -tuple $C \in \mathbb{C}^n$,
- $C(i)$: an index i denotes the i -th element of a chain (monotonically increasing sequence) $(C(m))_{m \in \mathbb{N}}$.

LEMMA D.21 (N-DIMENSIONAL DOMAIN). *Given the domain $(\mathbb{C}_{\gamma'}, \sqsubseteq_{\gamma'})$, we lift the equivalence relation, partial order, and the join operator of $\mathbb{C}_{\gamma'}$ to elements of $\mathbb{C}_{\gamma'}^n$ as follows:*

$$\begin{aligned} C \simeq_n \bar{C} &\iff \forall i \in I. C^i \simeq_{\gamma'} \bar{C}^i \\ C \sqsubseteq_n \bar{C} &\iff \forall i \in I. C^i \sqsubseteq_{\gamma'} \bar{C}^i \\ C \sqcup_n \bar{C} &:= \begin{pmatrix} C^1 \sqcup_{\gamma'} \bar{C}^1 \\ \vdots \\ C^n \sqcup_{\gamma'} \bar{C}^n \end{pmatrix} \end{aligned}$$

The proof that these satisfy the appropriate properties is trivial with the least element nle being the equivalence class of the n -tuple which holds while (true) skip in every component.

LEMMA D.22 (N-DIMENSIONAL CONTINUOUS G). *The function $G : \mathbb{C}_{\gamma'}^n \longrightarrow \mathbb{C}_{\gamma'}^n$, defined as*

$$G([C]) := [H(C)]$$

where

$$H(C) := \begin{pmatrix} h_1(C) \\ \vdots \\ h_n(C) \end{pmatrix}$$

and

$$h_i : \mathbb{C}^n \longrightarrow \mathbb{C}, \ h_i(C) := C_i[C, \gamma', F],$$

is continuous.

PROOF. Again, what needs to be proven is monotonicity and supremum-preservation.

Monotonicity. Analogously to the one-dimensional case, it is proven that h_i is monotonic for all $i \in I$. This implies that H and, therefore, G is monotonic as well.

Supremum-Preservation. Given a chain $(C(m))_{m \in \mathbb{N}}$, we need to prove:

$$\bigsqcup_{m \in \mathbb{N}} G([C(m)]) = G\left(\bigsqcup_{m \in \mathbb{N}} [C(m)]\right)$$

Due to the definition of G , we obtain for the left-hand side

$$\bigsqcup_{m \in \mathbb{N}} G([C(m)]) = \left[\bigsqcup_{m \in \mathbb{N}} H(C(m)) \right]$$

and for the right-hand side

$$G\left(\bigsqcup_{m \in \mathbb{N}} [C(m)]\right) = [H\left(\bigsqcup_{m \in \mathbb{N}} C(m)\right)]$$

It therefore suffices to show the equivalence of $\bigsqcup_{m \in \mathbb{N}} H(C(m))$ and $H(\bigsqcup_{m \in \mathbb{N}} C(m))$, i.e. for all $i \in I$:

$$\bigsqcup_{m \in \mathbb{N}} C_i[C(m), \gamma', F] \simeq_{\gamma'} C_i[\bigsqcup_{m \in \mathbb{N}} C(m) | m \in \mathbb{N}, \gamma', F]$$

This is proven analogously to the continuity in lemma D.16. \square

LEMMA D.23 (ADMISSIBLE SET). *Defining S_i^α analogous to lemma D.17 as*

$$S_i^\alpha := \{[\bar{C}] \in \mathbb{C}_{\gamma'} \mid \exists C \in [\bar{C}]. \forall t \in (\Gamma(\alpha))(f_i). \exists (P) (Q) \in \text{Int}_{\gamma', f_i}(t). \Gamma \models \{P\} C \{Q\}\}$$

then the set S^α , defined as

$$S^\alpha := \prod_{i \in I} S_i^\alpha$$

is an admissible subset of $(\mathbb{C}_{\gamma'}^n, \sqsubseteq_n)$.

PROOF. Least Element. The one-dimensional least element $\perp_{\gamma'}$, which is equivalent to the command while (true) skip, trivially semantically satisfies any over-approximating quadruple. We therefore have $\perp_n \in S^\alpha$.

Chain-Closure. Assume a chain $([\bar{C}](m))_{m \in \mathbb{N}} \subseteq S^\alpha$. Then, $([\bar{C}^i](m))_{m \in \mathbb{N}}$ is a chain in S_i^α for all $i \in I$. Since S_i^α is chain-closed (Lemma D.17), we have $\bigsqcup_{m \in \mathbb{N}} ([\bar{C}^i](m) | m \in \mathbb{N}) \in S_i^\alpha$. The desired result is then obtained by applying the definition of \bigsqcup . \square

LEMMA D.24 (N-SCOTT CONDITION). *Under the assumption that for all $i \in I$:*

$$\forall t \in (\Gamma(\alpha))(f_i). \exists t' \in \text{Int}_{\gamma', f_i}(t). \Gamma(\alpha) \vdash C_i : t'$$

and

$$\exists t' \in \text{Int}_{\gamma', f_i}((P^i(\alpha)) (Q^i(\alpha))). \Gamma(\alpha) \vdash C_i : t'$$

it holds that

$$G(S^\alpha) \subseteq S^\alpha$$

PROOF. Assume $[\bar{C}] \in S^\alpha$, i.e. $[\bar{C}^i] \in S_i^\alpha$, then for all $i \in I$ there exists a $C^i \in [\bar{C}^i]$ such that

$$\forall t \in (\Gamma(\alpha))(f_i). \exists (P) (Q) \in \text{Int}_{\gamma', f_i}(t). \Gamma \models \{P\} C^i \{Q\}$$

We aim to show $[H(C)] \in S^\alpha$, i.e. for all $i \in I$

$$[h_i(C)] \in S_i^\alpha$$

where $C := (C^1, \dots, C^n)$. What we prove is the general claim

$$\Gamma(\alpha) \vdash (P) C_i (Q) \implies \Gamma \models \{P\} C_i[C, \gamma', F] \{Q\}$$

which is shown by induction over the structure of C_i analogously to the proof of Lemma D.18. Then, instantiation with the existentially quantified t' yields the desired result. \square

LEMMA D.25 (AUXILIARY LEMMA). *For all $m \in \mathbb{N}$, $i \in I$ and $\sigma, \sigma' \in \text{State}$ it holds that*

$$\text{depth}_F(\sigma, C_i \Downarrow_{\gamma'} \sigma') \leq m \wedge \sigma, C_i \Downarrow_{\gamma'} \sigma' \iff \sigma, C_i[H^m(C), \gamma', F] \Downarrow_{\gamma'} \sigma'$$

This lemma is proven analogously to lemma D.20. We now proceed to the last step of the proof:

LEMMA D.26 (N-DIMENSIONAL LAST STEP). *It holds that*

$$\left(\begin{array}{c} C_1 \\ \vdots \\ C_n \end{array} \right) \in \text{lf}p(G)$$

PROOF. Given the known identities about least fixpoints and the definitions of G and H , we obtain:

$$lfp(G) = \bigsqcup_{m \in \mathbb{N}} G^m(\perp_n) = \bigsqcup_{m \in \mathbb{N}} [H^m(\text{wts}^n)] = [\bigsqcup_{m \in \mathbb{N}} H^m(\text{wts}^n)],$$

where wts^n denotes the n -tuple whose every component is `while (true) skip`. For this proof, given a vector v , we write $(v)_i$ to denote its i -th component. Assume $i \in I$ and $\sigma, \sigma' \in \text{State}$ such that $\sigma, C_i \Downarrow_{\gamma'} \sigma'$. This implies that the execution terminates and therefore there exists $m > 0$ such that $\text{depth}_F(\sigma, C_i \Downarrow_{\gamma'} \sigma') \leq m$.

Lemma D.25 then yields

$$\begin{aligned} & \sigma, C_i[H^{m-1}(\text{wts}^n)] \Downarrow_{\gamma'} \sigma' \\ \Rightarrow & \sigma, (H^m(\text{wts}^n))_i \Downarrow_{\gamma'} \sigma' \\ \Rightarrow & \sigma, \bigsqcup_{m \in \mathbb{N}} (H^m(\text{wts}^n))_i \Downarrow_{\gamma'} \sigma' \\ \Rightarrow & \sigma, \left(\bigsqcup_{m \in \mathbb{N}} H^m(\text{wts}^n) \right)_i \Downarrow_{\gamma'} \sigma' \end{aligned}$$

which concludes the proof. □

Theorem D.5 and Lemmas D.24 and D.26 then imply for all $i \in I$ that $[C_i] \in S_i^\alpha$.

E SOUNDNESS: ENVIRONMENT FORMATION

To prove Theorem 4.13, we first prove the following, slightly weaker, lemma:

LEMMA E.1. Assume an environment (γ, Γ) , a finite set of indices $I = \{1, \dots, n\}$ and let $\gamma' = \gamma[f_i \mapsto (\vec{x}_i, C_i, E_i) | i \in I]$, with $f_i \notin \text{dom}(\gamma)$ for all $i \in I$. Then, for any ordinal α , defining

$$\Gamma(\alpha) = \Gamma[f_i \mapsto \{(P^i(\beta)) (Q_i(\beta)) \mid \beta < \alpha\} \cup \{(P_\infty^i(\beta))(\text{False}) \mid \beta \leq \alpha\} \mid i \in I],$$

and assuming that

$$\forall i \in I, \alpha. \exists t' \in \text{Int}_{\gamma', f_i}((P^i(\alpha)) (Q^i(\alpha))). \Gamma(\alpha) \vdash C_i : t' \quad (4)$$

and

$$\forall i \in I, \alpha. \exists t' \in \text{Int}_{\gamma', f_i}((P_\infty^i(\alpha)) (\text{False})). \Gamma(\alpha) \vdash C_i : t' \quad (5)$$

it holds that

$$\models (\gamma, \Gamma) \implies (\forall \alpha. \models (\gamma', \Gamma(\alpha))).$$

PROOF. By transfinite induction on the ordinal α , reasoning about the zero, successor and limit-ordinal cases.

Zero case. When $\alpha = 0$, by definition we have $\Gamma(0) = \Gamma[f_i \mapsto \{(P_\infty^i(0))(\text{False})\} | i \in I]$. Let $f \in \text{dom}(\gamma')$: then, if $f \neq f_i, \forall i \in I$ and for any $t \in (\Gamma(0))(f)$, it holds that $t \in \Gamma(f)$ and from $\models (\gamma, \Gamma)$ we obtain the existence of a $t' \in \text{Int}_{\gamma', f}(t)$ such that $\models C : t'$.

Otherwise, $f = f_i$ for some $i \in I$, in which case $(\Gamma(0))(f)$ is a singleton set only holding the specification $t = (P_\infty^i(0)) (\text{False})$. We instantiate the n -dimensional Scott induction of [stoD.2](#) with $\alpha = 0$, which yields

$$\forall i \in I. [C_i] \in S_i^\alpha$$

that is, there exists a $C \in [C_i]$ such that

$$\Gamma \models (P_\infty^i(0)) C (\text{False})$$

The assumption $\models (\gamma, \Gamma)$ implies $\gamma \models (P_\infty^i(0)) C (\text{False})$. Therefore, C only calls on functions from $\text{dom}(\gamma)$, allowing us to arbitrarily extend the domain of γ , yielding $\gamma' \models (P_\infty^i(0)) C (\text{False})$, that is:

$$\begin{aligned} \forall \theta, s, h, h_f, o, s', h''. \theta, s, h \models P_\infty^i(0) \star \vec{z} = \text{null} &\implies \\ (s, h \uplus h_f), C \Downarrow_{\gamma'} o : (s', h'') &\implies \\ o \neq \text{miss} \wedge (\exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models \text{False}) & \end{aligned}$$

As C and C_i are behaviourally equivalent, we trivially obtain the same statement for C_i , i.e.

$$\gamma' \models (P_\infty^i(0)) C_i (\text{False})$$

which concludes this case of the proof.

Successor case. In the successor case, we assume the inductive hypothesis $(\text{IH}) \models (\gamma', \Gamma(\alpha))$ for an arbitrary ordinal α , and we need to prove that $\models (\gamma', \Gamma(\alpha + 1))$. By definition, $(\text{H1}) \text{dom}(\Gamma(\alpha)) = \text{dom}(\Gamma(\alpha + 1))$ and $(\text{H2}) (\Gamma(\alpha))(f) \subset (\Gamma(\alpha + 1))(f)$ holds for all f . Now, let f be in $\text{dom}(\Gamma(\alpha + 1))$. From the definition, we obtain that

$$(\Gamma(\alpha + 1))(f) = \begin{cases} \Gamma(f), & \text{for } f \neq f_i, \forall i \in I \\ (\Gamma(\alpha))(f_i) \cup \{(P^i(\alpha)) (Q^i(\alpha))\}, & \text{for } f = f_i, \text{ for some } i \in I \\ \cup \{(P_\infty^i(\alpha + 1)) (\text{False})\} \end{cases}$$

We prove the various cases separately. First, if $f \neq f_i \forall i \in I$, then the inductive hypothesis $\models (\gamma', \Gamma(\alpha))$ implies that

$$\forall f \in \text{dom}(\Gamma(\alpha)), t \in (\Gamma(\alpha))(f). f(\vec{x})\{C; \text{return } E\} \in \gamma' \implies \exists t' \in \text{Int}_{\gamma', f}(t). \gamma' \models C : t'$$

Since in this case $(\Gamma(\alpha + 1))(f) = \Gamma(f) = (\Gamma(\alpha))(f)$ and $\gamma(f) = \gamma'(f)$ since the domains of γ and Γ coincide, we immediately obtain the desired

$$\forall t \in (\Gamma(\alpha + 1))(f). \exists t' \in \text{Int}_{\gamma', f}(t). \gamma' \models C : t'.$$

Next, let $f = f_i$ for some $i \in I$ and let $t \in (\Gamma(\alpha))(f_i)$. Then, the inductive hypothesis $\models (\gamma', \Gamma(\alpha))$ immediately implies the existence of a $t' \in \text{Int}_{\gamma', f}(t)$ such that $\gamma' \models C_i : t'$.

Next, let $t = (P^i(\alpha)) (Q^i(\alpha))$. From (4), we know that there exists a $t' \in \text{Int}_{\gamma', f_i}(t)$ such that $\gamma' \vdash C_i : t'$. From there, given Theorem 4.12 and the inductive hypothesis $\models (\gamma', \Gamma(\alpha))$, we obtain that $\gamma' \models C_i : t'$.

Finally, let $t = (P_\infty^i(\alpha + 1)) (\text{False})$. Per definition, we have $\text{Int}_{\gamma', f_i}(t) = \{(P_\infty^i(\alpha + 1) \star \vec{z} = \text{null}) (\text{False})\}$, where $\vec{z} = \text{pv}(C_i) \setminus \{\vec{x}_i\}$. We instantiate the n -dimensional Scott induction from Appendix D.2 with $\alpha + 1$ and obtain

$$[C_i] \in s_i^{\alpha+1}$$

Analogously to the argumentation in the zero case, we obtain the desired

$$\gamma' \models (P_\infty^i(\alpha + 1)) C_i (\text{False})$$

concluding the successor case.

Limit Ordinal Case. Given an arbitrary limit ordinal α , we assume the inductive hypothesis (IH) $\forall \lambda < \alpha. \models (\gamma', \Gamma(\lambda))$, and our goal is to prove that $\models (\gamma', \Gamma(\alpha))$.

Let f be an arbitrary element from $\text{dom}(\Gamma(\alpha))$. First, if $f \neq f_i, \forall i \in I$, then we have $(\Gamma(\alpha))(f) = \Gamma(f)$ and $\gamma'(f) = \gamma(f)$. The reasoning for this case is the same as for the first part of the successor case. Otherwise, we have that $f = f_i$ for some $i \in I$. For this case, we first prove that the following two sets are equal:

$$\begin{aligned} A &= \{(P^i(\beta)) (Q^i(\beta)) \mid \beta < \alpha\} \cup \{(P_\infty^i(\beta)) (\text{False}) \mid \beta \leq \alpha\} \text{ and} \\ B &= \left(\bigcup_{\beta < \alpha} (\Gamma(\beta))(f_i) \right) \cup \{(P_\infty^i(\alpha)) (\text{False})\}. \end{aligned}$$

For the left-to-right inclusion, let $t \in A$.

Case 1. $t = (P^i(\lambda)) (Q^i(\lambda))$ for some $\lambda < \alpha$. As α is a limit ordinal, there exists another ordinal β such that $\lambda < \beta < \alpha$. By construction it holds that $t \in (\Gamma(\beta))(f_i)$ and hence $t \in B$.

Case 2. $t = (P_\infty^i(\lambda)) (\text{False})$ for some $\lambda < \alpha$. Per construction, we have $t \in (\Gamma(\lambda))(f_i)$ and therefore $t \in B$.

Case 3. $t = (P_\infty^i(\alpha)) (\text{False})$. Trivially, we have $t \in B$.

For the right-to-left inclusion, let $t \in B$.

Case 1. $t \in (\Gamma(\lambda))(f_i)$ for some $\lambda < \alpha$. Then, per construction, $t \in A$.

Case 2. $t = (P_\infty^i(\alpha)) (\text{False})$. Again, per construction, $t \in A$.

This yields the equality of A and B , which gives us that

$$(\Gamma(\alpha))(f_i) = \left(\bigcup_{\beta < \alpha} (\Gamma(\beta))(f_i) \right) \cup \{(P_\infty^i(\alpha)) (\text{False})\}.$$

Now, let $t \in (\Gamma(\alpha))(f_i)$, and consider the following two cases:

Case 1. $t \in (\Gamma(\lambda))(f_i)$ for some $\lambda < \alpha$. The inductive hypothesis implies $\models (\gamma', \Gamma(\lambda))$ and therefore that $\gamma' \models C_i : t'$ for some $t' \in \text{Int}_{\gamma', f_i}(t)$.

Case 2. $t = (P_\infty^i(\alpha)) (\text{False})$. This is proven by instantiating the n -dimensional Scott induction from Appendix D.2 with α , which analogously to the zero case, yields $\gamma' \models (P_\infty^i(\alpha)) C_i (\text{False})$, which concludes the proof. \square

Before the final soundness proof, we require one further lemma:

LEMMA E.2 (EXISTENTIALISATION). *Let $\models (\gamma, \Gamma), \gamma(f) = (\vec{x}, C_f, E')$, and let $X = \{(P(x)) (ok : Q_{ok}(x)) (err : Q_{err}(x)) \mid x \in O\}$. Then, if $X \subseteq \Gamma(f)$, it holds that $\models (\gamma, \bar{\Gamma})$, where*

$$\begin{aligned} \bar{\Gamma} = & \Gamma[f \mapsto (\Gamma(f)) \\ & \cup \{(\exists x. P(x) \star x \in O) (ok : \exists x. Q_{ok}(x) \star x \in O) (err : \exists x. Q_{err}(x) \star x \in O)\} \setminus X] \end{aligned}$$

PROOF. We need to prove that

$$\exists t \in \text{Int}_{\gamma, f}((\exists x. P(x) \star x \in O) (ok : \exists x. Q_{ok}(x) \star x \in O) (err : \exists x. Q_{err}(x) \star x \in O)). \gamma \models C_f : t$$

Over-approximation. Per construction, we know that the pre-condition of any internalisation of $(\exists x. P(x) \star x \in O) (ok : \exists x. Q_{ok}(x) \star x \in O) (err : \exists x. Q_{err}(x) \star x \in O)$ equals $(\exists x. P(x) \star x \in O) \star \vec{z} = \text{null}$,

where $\vec{z} = \text{pv}(C) \setminus \text{pv}(\exists x. P(x) \star x \in O)$, which is equivalent to $\exists x. P(x) \star x \in O \star \vec{z} = \text{null}$. We assume $\theta, s, h, s', h'', h_f, o$ such that

(O1) $\theta, s, h \models \exists x. P(x) \star x \in O \star \vec{z} = \text{null}$

(O2) $(s, h \uplus h_f), C_f \Downarrow_Y o : (s', h'')$

and aim to show:

$$o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, s', h' \models \bar{Q}_\epsilon$$

where \bar{Q}_ϵ is an internal postcondition of $(\exists x. P(x) \star x \in O) (ok : \exists x. Q_{ok}(x) \star x \in O) (err : \exists x. Q_{err}(x) \star x \in O)$.

- (O1) implies that $\exists v \in O. \theta[x \rightarrow v], s, h \models P(x) \star \vec{z} = \text{null}$.
- The assumptions of the lemma then imply that (O3) $o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta[x \rightarrow v], s', h' \models Q'_o$ for some Q'_o that is an internal post-condition of $(P(x)) (ok : Q_{ok}(x)) (err : Q_{err}(x))$, where $x \in O$, i.e. we have either

$$(O4a) \quad (o = ok) \wedge (Q_{ok}(x) \Leftrightarrow \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}])$$

$$(O4b) \quad (o = err) \wedge (Q_{err}(x) \Leftrightarrow \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}])$$

where, without loss of generality, we may assume that $x \notin \vec{p}$.

- (O3) implies that $\theta, s', h' \models \exists x. Q'_o \star x \in O$.
- To conclude the OX direction of the proof, we show that $\exists x. Q'_o \star x \in O$ is an internal post-condition of $(\exists x. P(x) \star x \in O) (ok : \exists x. Q_{ok}(x) \star x \in O) (err : \exists x. Q_{err}(x) \star x \in O)$ which is implied by (O4a) in case of successful, and by (O4b) in case of erroneous termination:

$$\begin{aligned} \exists x. Q_{ok}(x) \star x \in O &\Leftrightarrow \exists x. \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}] \star x \in O \\ &\Leftrightarrow \exists \vec{p}. \exists x. Q'_{ok}[\vec{p}/\vec{p}] \star x \in O \star \text{ret} = E'[\vec{p}/\vec{p}] \\ &\Leftrightarrow \exists \vec{p}. (\exists x. Q'_{ok}[\vec{p}/\vec{p}] \star x \in O) \star \text{ret} = E'[\vec{p}/\vec{p}] \\ &\Leftrightarrow \exists \vec{p}. (\exists x. Q'_{ok} \star x \in O)[\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}] \end{aligned}$$

$$\begin{aligned} \exists x. Q_{err}(x) \star x \in O &\Leftrightarrow \exists x. \exists \vec{p}. Q'_{err}[\vec{p}/\vec{p}] \star x \in O \\ &\Leftrightarrow \exists \vec{p}. \exists x. Q'_{err}[\vec{p}/\vec{p}] \star x \in O \\ &\Leftrightarrow \exists \vec{p}. (\exists x. Q'_{err} \star x \in O)[\vec{p}/\vec{p}] \end{aligned}$$

This concludes the OX direction.

Under-approximation. We assume θ, s', h', h_f, o such that $h' \# h_f$ and $\theta, s', h' \models \exists x. Q'_o \star x \in O$, where Q'_o is obtained from the OX case. This implies the existence of a $v \in O$ such that

$$\theta[x \rightarrow v], s', h' \models Q'_o$$

From the assumptions, we obtain the existence of s, h such that $\theta[x \rightarrow v], s, h \models P(x) \star \vec{z} = \text{null}$, which implies

$$\theta, s, h \models \exists x. P(x) \star \vec{z} = \text{null} \star x \in O$$

This concludes the proof as the last assertion is the (only) internal pre-condition of $(\exists x. P(x) \star x \in O) (ok : \exists x. Q_{ok}(x) \star x \in O) (err : \exists x. Q_{err}(x) \star x \in O)$.

□

With these lemmas, we can now easily prove theorem 4.13:

THEOREM 4.13. *Any well-formed environment is valid:*

$$\forall \gamma, \Gamma. \vdash (\gamma, \Gamma) \Longrightarrow \models (\gamma, \Gamma)$$

PROOF OF THEOREM 4.13. By induction on $\vdash (\gamma, \Gamma)$. When the last rule applied was the base rule for environments, we have that $(\gamma, \Gamma) = (\emptyset, \emptyset)$, meaning that $\text{dom}(\Gamma)$ is empty, and the statement to prove is trivially true. Otherwise, we assume the following hypotheses and inductive hypothesis:

- (H1) $\vdash (\gamma, \Gamma)$
 (H2) $\gamma' = \gamma[f_i \mapsto (\vec{x}_i, C_i, E_i) \mid i \in I]$, with $f_i \notin \text{dom}(\gamma)$ for all $i \in I$
 (H3) $\Gamma(\alpha) = \Gamma[f_i \mapsto \{(P^i(\beta)) (ok : Q_{ok}^i(\beta))(err : Q_{err}^i(\beta)) \mid \beta < \alpha\} \cup \{(P_\infty^i(\beta)) (\text{False}) \mid \beta \leq \alpha\}]_{i \in I}$
 (H4) $\forall i \in I, \alpha \in \mathcal{O}. \exists t \in \text{Int}_{\gamma', f_i}((P^i(\alpha)) (ok : Q_{ok}^i(\alpha))(err : Q_{err}^i(\alpha))). \Gamma(\alpha) \vdash C_i : t$
 (H5) $\forall i \in I, \alpha \in \mathcal{O}. \exists t \in \text{Int}_{\gamma', f_i}((P_\infty^i(\alpha)) (\text{False})). \Gamma(\alpha) \vdash C_i : t$
 (IH) $\models (\gamma, \Gamma)$

We will first prove that $\models (\gamma', \cup_\alpha \Gamma(\alpha))$, that is,

$$\begin{aligned} \text{dom}(\cup_\alpha \Gamma(\alpha)) &\subseteq \text{dom}(\gamma') \wedge \forall f, \vec{x}, C, E. f(\vec{x})\{C; \text{return } E\} \in \gamma' \\ &\Rightarrow (\forall t. t \in (\cup_\alpha \Gamma(\alpha))(f)) \\ &\Rightarrow \exists t' \in \text{Int}_{\gamma', f}(t). \gamma' \models C : t' \end{aligned}$$

where the notation $\cup_\alpha \Gamma(\alpha)$ denotes the function which maps f to the set $\cup_\alpha ((\Gamma(\alpha))(f))$. The first conjunct holds since Lemma E.1 implies that $\text{dom}(\Gamma(\alpha)) \subseteq \gamma'$ for all α . Now, assume f, \vec{x}, C, E, t such that

- (H6) $f(\vec{x})\{C; \text{return } E\} \in \gamma'$
 (H7) $t \in (\cup_\alpha \Gamma(\alpha))(f)$

By (H7), we have that there exists $\alpha' \in \mathcal{O}$ such that (H8) $t \in (\Gamma(\alpha'))(f)$. Then, from Lemma E.1 applied to (IH), (H2), (H4), and (H5), we obtain (H9) $\models (\gamma', \Gamma(\alpha'))$. By construction, we know that $\text{dom}(\cup_\alpha \Gamma(\alpha)) = \text{dom}(\Gamma(\alpha'))$, meaning that (H10) $f \in \text{dom}(\Gamma(\alpha'))$. Therefore, instantiating (H9) with (H10), (H8), and (H6), we obtain that there exists $t' \in \text{Int}_{\gamma', f}(t)$ such that $\models C : t'$, which implies $\models (\gamma', \cup_\alpha \Gamma(\alpha))$. Defining

- $P^i = \exists \alpha. P^i(\alpha) \star \alpha \in \mathcal{O}$
- $P_\infty^i = \exists \alpha. P_\infty^i(\alpha) \star \alpha \in \mathcal{O}$
- $Q_{ok}^i = \exists \alpha. Q_{ok}^i(\alpha) \star \alpha \in \mathcal{O}$
- $Q_{err}^i = \exists \alpha. Q_{err}^i(\alpha) \star \alpha \in \mathcal{O}$
- $\Gamma'' := \Gamma[f_i \mapsto \{(P^i) (ok : Q_{ok}^i)(err : Q_{err}^i), (P_\infty^i) (\text{False})\}]_{i \in I}$,

and applying Lemma E.2 twice to $\cup_\alpha \Gamma(\alpha)$ (once to the set of partially terminating specifications and once to the set of non-terminating specifications), we obtain $\models (\gamma', \Gamma'')$, concluding the soundness proof. \square

F FURTHER EXAMPLES

F.1 List Reverse

We consider an iterative implementation of list reversal, which takes a singly-linked list, reverses the pointers, and returns the pointer to the head of the new list. The code of the list reversal function is as follows:

$$\text{LRev}(x) \{ \text{while } (x \neq \text{null}) \{ z := [x + 1]; [x + 1] := y; y := x; x := z \}; \text{return } y \}$$

and the proof sketch of its correctness is given below. To verify this algorithm, we need to apply the iterative while rule: this means that we have to define the predicates P_i for all $i \in \mathbb{N}$:

$$P_i \triangleq \exists \beta, \gamma. \text{list}(y, \gamma) \star \text{list}(x, \beta) \star vs = \gamma^\dagger \cdot \beta \star i = |\gamma| \wedge R_i$$

where $R_i \triangleq (i = 0 \wedge x = x \wedge z = \text{null}) \vee (i > 0 \wedge x = z \wedge x \in \mathbb{N})$, and the minimal m for which P_i does not imply the loop condition is $|vs|$. In the proof of the while loop body, we use the equivalence (E1) $\text{list}(y, [v] \cdot \gamma) \Leftrightarrow \exists y. y \mapsto v, y \star \text{list}(y, \gamma)$ to add the currently processed node to the already reversed part of the list. Also, we inline the transition from the external to the internal pre-condition, as well as from the internal to the external post-condition, where $R \triangleq (|vs| = 0 \star x = \text{null}) \vee (|vs| > 0 \star x \in \mathbb{N})$.

```

 $\Gamma \vdash (x = x \star \text{list}(x, vs))$ 
LRev(x){
   $(x = x \star \text{list}(x, vs) \star y, z = \text{null})$ 
   $[[ \text{Establish } P_0 ]]$ 
   $(\exists \beta, \gamma. \text{list}(y, \gamma) \star \text{list}(x, \beta) \star vs = \gamma^\dagger \cdot \beta \star 0 = |\gamma| \star x = x \star z = \text{null})$ 
  while  $(x \neq \text{null})$  {
     $(P_i \star x \neq \text{null})$ 
     $(\exists \beta, \gamma. \text{list}(y, \gamma) \star \text{list}(x, \beta) \star vs = \gamma^\dagger \cdot \beta \star i = |\gamma| \star i > 0 \star x = z \star x \in \mathbb{N} \star x \neq \text{null})$ 
     $[[ \text{Unfold the list predicate, frame off } i > 0 \star x \in \mathbb{N} ]]$ 
     $(\exists \gamma, v, z, \beta'. \text{list}(y, \gamma) \star x \mapsto v, z \star \text{list}(z, \beta') \star vs = \gamma^\dagger \cdot [v] \cdot \beta' \star x = z \star i = |\gamma|)$ 
     $z := [x + 1];$ 
     $(\exists \gamma, v, z, \beta'. \text{list}(y, \gamma) \star x \mapsto v, z \star \text{list}(z, \beta') \star vs = \gamma^\dagger \cdot [v] \cdot \beta' \star z = z \star i = |\gamma|)$ 
     $[x + 1] := y;$ 
     $(\exists \gamma, v, z, \beta'. \text{list}(y, \gamma) \star x \mapsto v, y \star \text{list}(z, \beta') \star vs = \gamma^\dagger \cdot [v] \cdot \beta' \star z = z \star i = |\gamma|)$ 
     $y := x;$ 
     $(\exists \gamma, v, z, \beta'. y. y \mapsto v, y \star \text{list}(y, \gamma) \star \text{list}(z, \beta') \star vs = \gamma^\dagger \cdot [v] \cdot \beta' \star x = y \star z = z \star i = |\gamma|)$ 
     $[[ \text{Apply equivalence (E1)} ]]$ 
     $(\exists \gamma, v, z, \beta'. \text{list}(y, [v] \cdot \gamma) \star \text{list}(z, \beta') \star vs = ([v] \cdot \gamma)^\dagger \cdot \beta' \star x = y \star z = z \star i = |\gamma|)$ 
     $x := z;$ 
     $(\exists \gamma, v, z, \beta'. \text{list}(y, [v] \cdot \gamma) \star \text{list}(x, \beta') \star vs = ([v] \cdot \gamma)^\dagger \cdot \beta' \star x = z \star i = |\gamma|)$ 
     $[[ \text{Frame on } i > 0 \star x \in \mathbb{N} ]]$ 
     $(\exists \gamma, v, z, \beta'. \text{list}(y, [v] \cdot \gamma) \star \text{list}(x, \beta') \star vs = ([v] \cdot \gamma)^\dagger \cdot \beta' \star x = z \star i = |\gamma| \star i > 0 \star x \in \mathbb{N})$ 
     $[[ \text{Rename existentials: } v \cdot \gamma \rightarrow \gamma, \beta' \rightarrow \beta ]]$ 
     $(\exists \beta, \gamma. \text{list}(y, \gamma) \star \text{list}(x, \beta) \star vs = \gamma^\dagger \cdot \beta \star x = z \star i + 1 = |\gamma| \star i + 1 > 0 \star x \in \mathbb{N})$ 
     $(P_{i+1})$ 
  };
   $(\exists \beta, \gamma. \text{list}(y, \gamma) \star \text{list}(x, \beta) \star vs = \gamma^\dagger \cdot \beta \star x = z \star |vs| = |\gamma| \wedge R_{|vs|})$ 
   $(\text{list}(y, vs^\dagger) \star x, z = \text{null} \wedge R_{|vs|})$ 
  return y
   $[[ \text{Move to external post-condition, collapse existentials} ]]$ 
   $(\exists p_x, p_y, p_z. \text{list}(p_y, vs^\dagger) \star p_x, p_z = \text{null} \star \text{ret} = p_y \star R)$ 
   $(\text{list}(\text{ret}, vs^\dagger) \star R)$ 
}

```

(list(ret, vs[†]) ★ R)

F.2 List Free

We next consider the list-free algorithm, LFree(x), which frees all the nodes of a given singly-linked list starting at x. The algorithm is implemented as follows:

```

LFree(x){
  if (x = null) {
    r := null
  } else {
    y := x;
    x := [x + 1];
    free(y); free(y + 1);
    r := LFree(x)
  };
  return r
}

```

and the proof sketch of the body of the algorithm is given below. As the algorithm is recursive, the measure that we use is the length of the list, which corresponds to the number of pointers, $\alpha \triangleq |xs|$. As for the list length algorithm, we assume a valid environment (γ, Γ) , extend it with the LFree function, and construct $\Gamma(\alpha)$ appropriately, and doing the appropriate proof sketch for the function body:

```

 $\Gamma(\alpha) \vdash (x = x \star \text{list}(x, xs) \star \alpha = |xs| \star r, y = \text{null})$ 
  if (x = null) {
     $(x = x \star x = \text{null} \star \text{list}(x, xs) \star \alpha = |xs| \star r, y = \text{null})$ 
     $(x = x \star x = \text{null} \star xs = \epsilon \star \alpha = |xs| \star r, y = \text{null})$ 
    skip;
     $(x = x \star x = \text{null} \star xs = \epsilon \star \alpha = |xs| \star r, y = \text{null})$ 
  } else {
     $(x = x \star x \neq \text{null} \star \text{list}(x, xs) \star \alpha = |xs| \star r, y = \text{null})$ 
     $(\exists x', v, xs'. x = x \star x \mapsto v, x' \star \text{list}(x', xs') \star xs = x : xs' \star \alpha = |xs| \star r, y = \text{null})$ 
    y := x;
     $(\exists x', v, xs'. x = x \star y = x \star x \mapsto v, x' \star \text{list}(x', xs') \star xs = x : xs' \star \alpha = |xs| \star r = \text{null})$ 
    x := [x + 1];
     $(\exists x', v, xs'. x = x' \star y = x \star x \mapsto v, x' \star \text{list}(x', xs') \star xs = x : xs' \star \alpha = |xs| \star r = \text{null})$ 
     $(\exists x', v, xs'. x = x' \star y = x \star y \mapsto v, x' \star \text{list}(x', xs') \star xs = x : xs' \star \alpha = |xs| \star r = \text{null})$ 
    free(y);
     $(\exists x', xs'. x = x' \star y = x \star y \mapsto \emptyset, x' \star \text{list}(x', xs') \star xs = x : xs' \star \alpha - 1 = |xs'| \star r = \text{null})$ 
    free(y + 1);
    [ as  $\alpha - 1 < \alpha$ , we can apply the specification for  $\alpha - 1$  ]
     $(\exists x', xs'. x = x' \star y = x \star y \mapsto \emptyset, \emptyset \star \text{list}(x', xs') \star xs = x : xs' \star \alpha - 1 = |xs'| \star r = \text{null})$ 
    r := ListDispose(x)
     $(\exists x', xs'. x = x' \star y = x \star x \mapsto \emptyset, \emptyset \star \text{freed}(x' : xs') \star xs = x : xs' \star \alpha' = |xs'| \star r = \text{null})$ 
  };
   $\left( (x = x \star x = \text{null} \star xs = \epsilon \star r, y = \text{null} \star \alpha = |xs|) \vee \right.$ 
   $\left. (\exists x', xs'. x = x' \star y = x \star x \mapsto \emptyset, \emptyset \star \text{freed}(x' : xs') \star xs = x : xs' \star r = \text{null} \star \alpha = |xs|) \right)$ 

```

We conclude the proof by moving from the internal to the external specification:

$$\begin{aligned}
& \exists p_x, p_y, p_r. \text{ret} = p_r \star (p_x = x \star x = \text{null} \star xs = \epsilon \star p_r, p_y = \text{null} \star \alpha = |xs|) \vee \\
& \quad (\exists x', xs'. p_x = x' \star p_y = x \star x \mapsto \emptyset, \emptyset \star \text{freed}(x' : xs') \star xs = x : xs' \star \\
& \quad \quad p_r = \text{null} \star \alpha = |xs|) \\
\Leftrightarrow & \text{ret} = \text{null} \star \alpha = |xs| \star ((x = \text{null} \star xs = \epsilon) \vee \\
& \quad (\exists x', xs'. x \mapsto \emptyset, \emptyset \star \text{freed}(x' : xs') \star xs = x : xs')) \\
\Leftrightarrow & \text{freed}(x : xs) \star \text{ret} = \text{null} \star \alpha = |xs|
\end{aligned}$$

F.3 List Algorithm Client

We consider the following client of our three list algorithms

```

LClient(x) {
  l := LLen(x);
  if (l < 5) { r := LFree(x); error("LTS") } else {
    if (l > 10) { while (true) { skip } } else {
      r := ListReverse(l)
    }
  }
};
return r
}

```

and prove that it satisfies the following ESL specification:

$$\begin{aligned}
& (x = x \star \text{list}(x, vs)) \\
& \text{LClient}(x) \\
& (ok : 5 \leq |vs| \leq 10 \star \text{list}(\text{ret}, vs^\dagger) \star R) \\
& (err : |vs| < 5 \star (\exists xs. \text{freed}(x : xs) \star |xs| = |vs|) \star err = \text{"LTS"})
\end{aligned}$$

We prove the three branches separately, exposing the non-terminating case, and then join the obtained specifications through the admissible disjunction property, to obtain the above specification. We give two of the three proof sketches below; the third is analogous to the first. We denote the above success post-condition by Q_{ok} , the above faulting post-condition by Q_{err} , and assume a specification context Γ that has the appropriate specifications of the called functions.

$$\begin{aligned}
& \Gamma \vdash (x = x \star \text{list}(x, vs) \star |vs| < 5) \\
& \text{LClient}(x) \{ \\
& \quad (x = x \star \text{list}(x, vs) \star |vs| < 5 \star l, r = \text{null}) \\
& \quad l := \text{LLen}(x); \\
& \quad (x = x \star \text{list}(x, vs) \star |vs| < 5 \star l = |vs| \star r = \text{null}) \\
& \quad \text{if } (l < 5) \{ \\
& \quad \quad (x = x \star \text{list}(x, vs) \star |vs| < 5 \star l = |vs| \star r = \text{null}) \\
& \quad \quad r := \text{LFree}(x); \\
& \quad \quad (x = x \star (\exists xs. \text{freed}(x : xs) \star |xs| = |vs|) \star |vs| < 5 \star l = |vs| \star r = \text{null}) \\
& \quad \quad \text{error}(\text{"LTS"}) \\
& \quad \quad (err : Q_{err} \star x = x \star l = |vs| \star r = \text{null}) \\
& \quad \} \text{ else } \{ \\
& \quad \quad (\text{False}) \dots (\text{False}) \\
& \quad \}; \\
& \quad (err : Q_{err} \star x = x \star l = |vs| \star r = \text{null}) \\
& \quad \text{return } r \\
& \quad (err : Q_{err} \star (\exists a, b, c. a = x \star b = |vs| \star c = \text{null})) \\
& \quad (err : Q_{err}) \\
& \}
\end{aligned}$$

```

( err : Q_err )

Γ ⊢ ( x = x ★ list(x, vs) ★ |vs| > 10 )
  LClient(x) {
    ( x = x ★ list(x, vs) ★ |vs| > 10 ★ l, r = null )
    l := LLen(x);
    ( x = x ★ list(x, vs) ★ |vs| > 10 ★ l = |vs| ★ r = null )
    if (l < 5) {
      ( False ) ... ( False )
    } else {
      ( x = x ★ list(x, vs) ★ |vs| > 10 ★ l = |vs| ★ r = null )
      if (l > 10) {
        ( x = x ★ list(x, vs) ★ |vs| > 10 ★ l = |vs| ★ r = null )
        while (true) {skip}
        ( False )
      } else {
        ( False ) ... ( False )
      }
    };
    }; ( False )
    return r
    ( False )
  } ( False )

```

The three obtained specifications then yield via disjunction:

$$\left(\begin{array}{l} (x = x \star \text{list}(x, \text{vs}) \star |\text{vs}| < 5) \vee \\ (x = x \star \text{list}(x, \text{vs}) \star 5 \leq |\text{vs}| \leq 10) \vee \\ (x = x \star \text{list}(x, \text{vs}) \star |\text{vs}| > 10) \end{array} \right) \text{LClient}(x) \text{ (ok : False } \vee Q_{ok} \vee \text{False) (err : False } \vee Q_{err} \vee \text{False)}$$

and via equivalence the desired

$$(x = x \star \text{list}(x, \text{vs})) \text{LClient}(x) \text{ (ok : } Q_{ok} \text{) (err : } Q_{err} \text{)}$$

F.4 Mutual Recursion: even/odd

In addition to reasoning about recursive functions, ESL allows us to reason about mutually recursive function as well. We illustrate this by using a simple example consisting of two functions which determine whether a natural number is even or odd, whose implementations are given below.

<pre> isEven(n) { if (n = 0) { b := true } else { n := n - 1; b := isOdd(n) }; return b } </pre>	<pre> isOdd(n) { if (n = 0) { b := false } else { n := n - 1; b := isEven(n) }; return b } </pre>
--	---

To reason about these two functions, we introduce two (also mutually recursive) predicates:

$$\begin{aligned} \text{even}(n) &\triangleq n = 0 \vee \text{odd}(n - 1) \\ \text{odd}(n) &\triangleq n = 1 \vee \text{even}(n - 1). \end{aligned}$$

and give the pre-condition and the external ($Q_{\text{even}}(\alpha)$ and $Q_{\text{odd}}(\alpha)$) and internal ($Q'_{\text{even}}(\alpha)$ and $Q'_{\text{odd}}(\alpha)$) post-conditions for the two functions, noting that both share the same pre-condition $P(\alpha)$, and that we again

use $\alpha = n$ for the decreasing measure, just as in the list length case:

$$\begin{aligned}
P(\alpha) &\triangleq n = n \star n \in \mathbb{N} \star n = \alpha \\
Q_{\text{even}}(\alpha) &\triangleq (\text{ret} = \text{false} \star \text{odd}(n) \star n = \alpha) \vee (\text{ret} = \text{true} \star \text{even}(n) \star n = \alpha) \\
Q_{\text{odd}}(\alpha) &\triangleq (\text{ret} = \text{false} \star \text{even}(n) \star n = \alpha) \vee (\text{ret} = \text{true} \star \text{odd}(n) \star n = \alpha) \\
Q'_{\text{even}}(\alpha) &\triangleq (n = n \star n = 0 \star b = \text{true} \star n = \alpha) \\
&\quad \vee (n = n - 1 \star b = \text{false} \star \text{odd}(n) \star n = \alpha) \\
&\quad \vee (n = n - 1 \star b = \text{true} \star \text{even}(n) \star n = \alpha) \\
Q'_{\text{odd}}(\alpha) &\triangleq (n = n \star n = 0 \star b = \text{false} \star n = \alpha) \\
&\quad \vee (n = n - 1 \star b = \text{true} \star \text{odd}(n) \star n = \alpha) \\
&\quad \vee (n = n - 1 \star b = \text{false} \star \text{even}(n) \star n = \alpha)
\end{aligned}$$

We further assume a well-formed environment (γ, Γ) such that $\text{isEven}, \text{isOdd} \notin \text{dom}(\gamma)$ and extend it as follows:

$$\begin{aligned}
\gamma' &\triangleq \gamma[\text{isEven} \mapsto (\{n\}, C_{\text{even}}, b), \text{isOdd} \mapsto (\{n\}, C_{\text{odd}}, b)] \\
\Gamma(\alpha) &\triangleq \Gamma[\text{isEven} \mapsto \{(P(\beta)) (ok : Q_{\text{even}}(\beta)) \mid \beta < \alpha\}, \\
&\quad \text{isOdd} \mapsto \{(P(\beta)) (ok : Q_{\text{odd}}(\beta)) \mid \beta < \alpha\}]
\end{aligned}$$

where C_{even} and C_{odd} denote the appropriate function bodies. Our goal is to prove

$$\begin{aligned}
\Gamma(\alpha) &\vdash (P(\alpha)) \text{ } C_{\text{even}} \text{ } (ok : Q'_{\text{even}}(\alpha)) \\
\Gamma(\alpha) &\vdash (P(\alpha)) \text{ } C_{\text{odd}} \text{ } (ok : Q'_{\text{odd}}(\alpha))
\end{aligned}$$

which we do in the proof sketches given below, first for isEven and then for isOdd :

$$\begin{aligned}
&\Gamma(\alpha) \vdash (P(\alpha) \star b = \text{null}) \\
&\quad (n = n \star n \in \mathbb{N} \star n = \alpha \star b = \text{null}) \\
&\quad \text{if } (n = 0) \{ \\
&\quad \quad (n = n \star n = 0 \star n = \alpha \star b = \text{null}) \\
&\quad \quad b := \text{true} \\
&\quad \quad (n = n \star n = 0 \star n = \alpha \star b = \text{true}) \\
&\quad \} \text{ else } \{ \\
&\quad \quad (n = n \star n > 0 \star n = \alpha \star b = \text{null}) \\
&\quad \quad n := n - 1 \\
&\quad \quad (n = n - 1 \star n > 0 \star n = \alpha \star b = \text{null}) \\
&\quad \quad (n = n - 1 \star n - 1 \in \mathbb{N} \star n - 1 = \alpha - 1 \star b = \text{null}) \\
&\quad \quad [\text{as } \alpha - 1 < \alpha, \text{ we can apply the specification for } \alpha - 1] \\
&\quad \quad b := \text{isOdd}(n) \\
&\quad \quad (n = n - 1 \star \exists b. ((b = \text{false} \star \text{even}(n - 1)) \vee (b = \text{true} \star \text{odd}(n - 1))) \star n = \alpha \star b = b) \\
&\quad \quad (n = n - 1 \star ((b = \text{false} \star \text{odd}(n) \star n = \alpha) \vee (b = \text{true} \star \text{even}(n) \star n = \alpha))) \\
&\quad \quad (n = n - 1 \star b = \text{false} \star \text{odd}(n) \star n = \alpha) \vee (n = n - 1 \star b = \text{true} \star \text{even}(n) \star n = \alpha) \\
&\quad \} \\
&\quad \left((n = n \star n = 0 \star b = \text{true} \star n = \alpha) \vee (n = n - 1 \star b = \text{false} \star \text{odd}(n) \star n = \alpha) \vee \right. \\
&\quad \left. (n = n - 1 \star b = \text{true} \star \text{even}(n) \star n = \alpha) \right)
\end{aligned}$$

$$\begin{aligned}
& \Gamma(\alpha) \vdash (P(\alpha) \star b = \text{null}) \\
& \quad (n = n \star n \in \mathbb{N} \star n = \alpha \star b = \text{null}) \\
& \quad \text{if } (n = 0) \{ \\
& \quad \quad (n = n \star n = 0 \star n = \alpha \star b = \text{null}) \\
& \quad \quad b := \text{false} \\
& \quad \quad (n = n \star n = 0 \star n = \alpha \star b = \text{false}) \\
& \quad \quad \} \text{ else } \{ \\
& \quad \quad (n = n \star n > 0 \star n = \alpha \star b = \text{null}) \\
& \quad \quad n := n - 1 \\
& \quad \quad (n = n - 1 \star n > 0 \star n = \alpha \star b = \text{null}) \\
& \quad \quad (n = n - 1 \star n - 1 \in \mathbb{N} \star n - 1 = \alpha - 1 \star b = \text{null}) \\
& \quad \quad \text{[since } \alpha - 1 < \alpha, \text{ we can apply the specification for } \alpha - 1\text{]} \\
& \quad \quad b := \text{isEven}(n) \\
& \quad \quad (n = n - 1 \star \exists b. ((b = \text{false} \star \text{odd}(n - 1)) \vee (b = \text{true} \star \text{even}(n - 1))) \star n = \alpha \star b = b) \\
& \quad \quad (n = n - 1 \star ((b = \text{false} \star \text{even}(n) \star n = \alpha) \vee (b = \text{true} \star \text{odd}(n) \star n = \alpha))) \\
& \quad \quad ((n = n - 1 \star b = \text{false} \star \text{even}(n) \star n = \alpha) \vee (n = n - 1 \star b = \text{true} \star \text{odd}(n) \star n = \alpha)) \\
& \quad \quad \} \\
& \quad \left((n = n \star n = 0 \star n = \alpha \star b = \text{false}) \vee (n = n - 1 \star b = \text{false} \star \text{even}(n) \star n = \alpha) \vee \right. \\
& \quad \left. (n = n - 1 \star b = \text{true} \star \text{odd}(n) \star n = \alpha) \right)
\end{aligned}$$

To complete the proof, we need to show that $Q_i \Leftrightarrow \exists \vec{p}. Q'_i[\vec{p}/\vec{p}] \star \text{ret} = b[\vec{p}/\vec{p}]$ for $i \in \{\text{even}, \text{odd}\}$. As the two cases are analogous, we only show the even case in detail:

$$\begin{aligned}
& \exists \vec{p}. Q'_{\text{even}}(\alpha)[\vec{p}/\vec{p}] \star \text{ret} = E[\vec{p}/\vec{p}] \\
& \quad ((p_n = n \star n = 0 \star p_b = \text{true} \star n = \alpha) \vee \\
& \Leftrightarrow \exists p_n, p_b. (p_n = n - 1 \star p_b = \text{false} \star \text{odd}(n) \star n = \alpha) \vee \\
& \quad (p_n = n - 1 \star p_b = \text{true} \star \text{even}(n) \star n = \alpha)) \star \text{ret} = p_b \\
& \Leftrightarrow (n = 0 \star \text{ret} = \text{true} \star \text{even}(n) \star n = \alpha) \vee (n \dot{>} 0 \star \text{ret} = \text{false} \star \text{odd}(n) \star n = \alpha) \vee \\
& \quad (n \dot{>} 0 \star \text{ret} = \text{true} \star \text{even}(n) \star n = \alpha) \\
& \Leftrightarrow (n \dot{>} 0 \star \text{ret} = \text{false} \star \text{odd}(n) \star n = \alpha) \vee (n \in \mathbb{N} \star \text{ret} = \text{true} \star \text{even}(n) \star n = \alpha) \\
& \Leftrightarrow (\text{ret} = \text{false} \star \text{odd}(n) \star n = \alpha) \vee (\text{ret} = \text{true} \star \text{even}(n) \star n = \alpha) \\
& \Leftrightarrow Q_{\text{even}}(\alpha)
\end{aligned}$$

F.5 More Complex Mutual Recursion: even/odd/list length

In the previous examples featuring recursion and mutual recursion, the choice of the measure is straightforward. For list length and list disposal, we traverse a non-cyclic list, therefore decreasing the distance to the end of the list in every step. For even/odd, each function decreases the function argument before passing it on to the other function, therefore also creating a natural measure.

In the real world, however, we might come across clusters of mutually recursive functions where not every function reduces the obvious measure (e.g., wrapper functions). As long as any function call terminates, we can still reason about such clusters by defining an appropriate measure. To illustrate this, we will look at a

collection of three functions, which compute the length of a list in a convoluted, mutually-recursive way:

<pre> LL(x) { if (x = null) { r := 0; } else { v := [x]; if (even(v)) { r := g(x); } else { r := f(x); }; }; return r } </pre>	<pre> f(x) { v := [x]; if (even(v)) { x := [x + 1]; r := LL(x); r := r + 1; } else { r := g(x); }; return r } </pre>	<pre> g(x) { v := [x]; if (odd(v)) { x := [x + 1]; r := LL(x); r := r + 1; } else { r := f(x); }; return r } </pre>
--	--	---

Intuitively, whenever either of the functions is called with an argument x , which is the head of a (non-cyclic) list, it computes the length of the list. The LL function calls g if the first value of the list is even and f otherwise. The function g , however, does the same test and calls f if the input was even. Otherwise, it moves one element down the list and calls LL on the now shortened list. The function f moves down the list by one element and calls LL on the shortened list if the first value is even, and calls g on the initial list, if not.

As the functions branch on whether or not values of the list are divisible by 2, we adjust the $list(x, vs)$ predicate slightly to include the condition that vs is a list of natural numbers:

$$list_{\mathbb{N}}(x, vs) \triangleq (x = \text{null} \star vs = \epsilon) \vee (\exists v, x, vs'. x \mapsto v, x' \star v \in \mathbb{N} \star list_{\mathbb{N}}(x', vs') \star vs = v : vs')$$

and furthermore require a trivial property of the previously introduced $even()$ and $odd()$ predicates, stating that $even(v) \vee odd(v) \Leftrightarrow v \in \mathbb{N}$. Assuming a valid environment $\vdash (\gamma, \Gamma)$, we extend it as follows

$$\begin{aligned}
\gamma' &\triangleq \gamma[LL \mapsto (\{x\}, C_{LL}, r), \quad f \mapsto (\{x\}, C_f, r), \quad g \mapsto (\{x\}, C_g, r)] \\
\Gamma(\alpha) &\triangleq \Gamma[LL \mapsto \{(P_{LL}(\beta))(ok : Q_{LL}) | \beta < \alpha\}, \\
&\quad f \mapsto \{(P_f(\beta))(ok : Q_f) | \beta < \alpha\}, g \mapsto \{(P_g(\beta))(ok : Q_g) | \beta < \alpha\}]
\end{aligned}$$

Furthermore, we define

$$\begin{aligned}
\Gamma'' &\triangleq \Gamma[LL \mapsto \{(list_{\mathbb{N}}(x, vs) \star x = x \star (3|vs| + 2) \in O) \\
&\quad (ok : list_{\mathbb{N}}(x, vs) \star ret = vs \star (3|vs| + 2) \in O)\}, \\
f &\mapsto \{(list_{\mathbb{N}}(x, v : vs') \star x = x \star 3|v : vs'| + (v \bmod 2) \in O) \\
&\quad (ok : list_{\mathbb{N}}(x, v : vs') \star ret = |v : vs'| \star 3|v : vs'| + (v \bmod 2)) \in O\}, \\
g &\mapsto \{(list_{\mathbb{N}}(x, v : vs') \star x = x \star 3|v : vs'| + 1 - (v \bmod 2) \in O) \\
&\quad (ok : list_{\mathbb{N}}(x, v : vs') \star ret = |v : vs'| \star 3|v : vs'| + 1 - (v \bmod 2) \in O)\}]
\end{aligned}$$

and wish to prove $\vdash (\gamma', \Gamma'')$. To this end, we prove the following three specifications:

$$\begin{aligned}
\Gamma(\alpha) &\vdash (P_{LL}(\alpha) \star r, v = \text{null}) C_{LL} (ok : Q'_{LL}(\alpha)) \\
\Gamma(\alpha) &\vdash (P_f(\alpha) \star r, v = \text{null}) C_f (ok : Q'_f(\alpha)) \\
\Gamma(\alpha) &\vdash (P_g(\alpha) \star r, v = \text{null}) C_g (ok : Q'_g(\alpha))
\end{aligned}$$

where C_{LL} , C_f , and C_g denote the appropriate function bodies, and the function pre-conditions, capturing the function pre-conditions (with and without measure) and post-conditions (internal and external), are defined

as follows:

$$\begin{aligned}
P_{LL}(\alpha) &= \text{list}_{\mathbb{N}}(x, vs) \star x = x \star \alpha = 3|vs| + 2 \\
P_f(\alpha) &= \text{list}_{\mathbb{N}}(x, v : vs') \star x = x \star \alpha = 3|v : vs'| + (v \bmod 2) \\
P_g(\alpha) &= \text{list}_{\mathbb{N}}(x, v : vs') \star x = x \star \alpha = 3|v : vs'| + 1 - (v \bmod 2) \\
Q'_{LL}(\alpha) &= (x = \text{null} \star vs = \epsilon \star x = x \star r = |vs| \star v = \text{null} \star \alpha = 3|vs| + 2) \vee \\
&\quad (\exists v, vs'. \text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star r = |v : vs'| \star \alpha = 3|vs| + 2) \\
Q'_f(\alpha) &= \exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star v = v \star r = |v : vs'| \\
&\quad \star (x = x' \star \text{even}(v) \vee x = x \star \text{odd}(v)) \star \alpha = 3|v : vs'| + (v \bmod 2) \\
Q'_g(\alpha) &= \exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star v = v \star r = |v : vs'| \\
&\quad \star (x = x' \star \text{odd}(v) \vee x = x \star \text{even}(v)) \star \alpha = 3|v : vs'| + 1 - (v \bmod 2) \\
Q_{LL}(\alpha) &= \text{list}_{\mathbb{N}}(x, vs) \star \text{ret} = |vs| \star \alpha = 3|vs| + 2 \\
Q_f(\alpha) &= \text{list}_{\mathbb{N}}(x, v : vs') \star \text{ret} = |v : vs'| \star \alpha = 3|v : vs'| + (v \bmod 2) \\
Q_g(\alpha) &= \text{list}_{\mathbb{N}}(x, v : vs') \star \text{ret} = |v : vs'| \star \alpha = 3|v : vs'| + 1 - (v \bmod 2)
\end{aligned}$$

We give the proof sketch for list length and f below. The proof sketch for g is analogous to that of f.

$$\begin{aligned}
&\Gamma(\alpha) \vdash (\text{list}_{\mathbb{N}}(x, vs) \star x = x \star \alpha = 3|vs| + 2 \star r, v = \text{null}) \\
&\quad \text{if } (x = \text{null}) \{ \\
&\quad \quad (\text{list}_{\mathbb{N}}(x, vs) \star x = x \star \alpha = 3|vs| + 2 \star r, v = \text{null}) \\
&\quad \quad r := 0; \\
&\quad \quad (\text{list}_{\mathbb{N}}(x, vs) \star x = x \star \alpha = 3|vs| + 2 \star r = 0 \star x, v = \text{null}) \\
&\quad \quad (x = \text{null} \star vs = \epsilon \star x = x \star r = |vs| \star \alpha = 3|vs| + 2 \star v = \text{null}) \\
&\quad \} \text{ else } \{ \\
&\quad \quad (\exists v, x', vs'. x \mapsto v, x' \star v \in \mathbb{N} \star \text{list}_{\mathbb{N}}(x', vs') \star vs = v : vs' \star x = x \star \alpha = 3|vs| + 2 \star r, v = \text{null}) \\
&\quad \quad v := [x]; \\
&\quad \quad \left(\begin{array}{l} \exists v, x', vs'. x \mapsto v, x' \star v \in \mathbb{N} \star \text{list}_{\mathbb{N}}(x', vs') \star vs = v : vs' \star \\ x = x \star v = v \star \alpha = 3|vs| + 2 \star r = \text{null} \end{array} \right) \\
&\quad \quad (\exists v, vs'. \text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star \alpha = 3|vs| + 2 \star r = \text{null}) \\
&\quad \quad \left(\begin{array}{l} (\text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star v \in \mathbb{N} \star \alpha = 3|vs| + 2 \star r = \text{null}) \\ \text{if } (\text{even}(v)) \{ \\ \quad \left(\begin{array}{l} \text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star \text{even}(v) \\ \star \alpha - 1 = 3|v : vs'| + 1 - (v \bmod 2) \star r = \text{null} \end{array} \right) \\ \quad \text{[[as } \alpha - 1 < \alpha, \text{ we can apply } g\text{'s specification for } \alpha - 1\text{]]} \\ \quad \text{fr, eq } \left(\begin{array}{l} (r = \text{null} \star x = x \star \text{list}_{\mathbb{N}}(x, v : vs') \star \alpha - 1 = 3|v : vs'| + 1 - (v \bmod 2)) \\ r := g(x); \\ (x = x \star \text{list}_{\mathbb{N}}(x, v : vs') \star r = |v : vs'| \star \alpha - 1 = 3|v : vs'| + 1 - (v \bmod 2)) \end{array} \right) \\ \quad (\text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star \text{even}(v) \star r = |vs| \star \alpha = 3|vs| + 2) \\ \quad \} \text{ else } \{ \\ \quad \quad \left(\begin{array}{l} \text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star \\ \text{odd}(v) \star r = \text{null} \star \alpha - 1 = 3|vs| + (v \bmod 2) \end{array} \right) \\ \quad \quad \text{[[as } \alpha - 1 < \alpha, \text{ we can apply } f\text{'s specification for } \alpha - 1\text{]]} \\ \quad \quad \text{fr, eq } \left(\begin{array}{l} (\text{list}_{\mathbb{N}}(x, v : vs') \star x = x \star r = \text{null} \star \alpha - 1 = 3|vs| + (v \bmod 2)) \\ r := f(x); \\ (\text{list}_{\mathbb{N}}(x, v : vs') \star x = x \star r = |v : vs'| \star \alpha - 1 = 3|vs| + (v \bmod 2)) \end{array} \right) \\ \quad \quad (\text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star \text{odd}(v) \star r = |vs| \star \alpha = 3|vs| + 2) \\ \quad \quad \} \\ \quad \quad \left(\text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star r = |vs| \star (\text{even}(v) \vee \text{odd}(v)) \star \alpha = 3|vs| + 2 \right) \\ \quad \quad (\exists v \in \mathbb{N}, vs'. \text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star x = x \star v = v \star r = |vs| \star \alpha = 3|vs| + 2) \\ \quad \} \\
\end{aligned}$$

$$\left((x = \text{null} \star vs = \epsilon \star x = x \star r = |vs| \star v = \text{null} \star \alpha = 3|vs| + 2) \vee \right. \\ \left. (\exists v \in \mathbb{N}, vs'. \text{list}_{\mathbb{N}}(x, v : vs')) \star vs = v : vs' \star x = x \star v = v \star r = |vs| \star \alpha = 3|vs| + 2) \right)$$

$$\begin{aligned} \Gamma(\alpha) \vdash & (\text{list}_{\mathbb{N}}(x, v : vs') \star x = x \star \alpha = 3|v : vs'| + (v \bmod 2) \star v, r = \text{null}) \\ & (\exists x'. x \mapsto v, x' \star v \in \mathbb{N} \star \text{list}_{\mathbb{N}}(x', vs') \star x = x \star \alpha = 3|v : vs'| + (v \bmod 2) \star v, r = \text{null}) \\ & v := [x]; \\ & (\exists x'. x \mapsto v, x' \star v \in \mathbb{N} \star \text{list}_{\mathbb{N}}(x', vs') \star x = x \star v = v \star \alpha = 3|v : vs'| + (v \bmod 2) \star r = \text{null}) \\ & (\exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star x = x \star v = v \star v \in \mathbb{N} \star \alpha = 3|v : vs'| + (v \bmod 2) \star r = \text{null}) \\ & \text{if } (\text{even}(v)) \{ \\ & \quad (\exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star x = x \star v = v \star \text{even}(v) \star \alpha = 3|v : vs'| \star r = \text{null}) \\ & \quad x := [x + 1]; \\ & \quad (\exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star x = x' \star v = v \star \text{even}(v) \star \alpha - 1 = 3|vs'| + 2 \star r = \text{null}) \\ & \quad \text{[as } \alpha - 1 < \alpha, \text{ we can apply LL's specifications for } \alpha - 1 \text{]} \\ & \quad \text{fr + ex} \left| \begin{array}{l} (\text{list}_{\mathbb{N}}(x', vs') \star x = x' \star r = \text{null} \star \alpha - 1 = 3|vs'| + 2) \\ r := \text{LL}(x); \\ (\text{list}_{\mathbb{N}}(x', vs') \star x = x' \star r = |vs'| \star \alpha - 1 = 3|vs'| + 2) \\ r := r + 1; \\ (\text{list}_{\mathbb{N}}(x', vs') \star x = x' \star r = |vs'| + 1 \star \alpha - 1 = 3|vs'| + 2) \end{array} \right. \\ & \quad \left(\exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star x = x' \star v = v \star \right. \\ & \quad \left. r = |v : vs'| \star \text{even}(v) \star \alpha = 3|v : vs'| + (v \bmod 2) \right) \\ & \quad \} \text{ else } \{ \\ & \quad \left(\exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star x = x \star v = v \star \right. \\ & \quad \left. \text{odd}(v) \star \alpha - 1 = 3|v : vs'| + 1 - (v \bmod 2) \star r = \text{null} \right) \\ & \quad \text{[as } \alpha - 1 < \alpha, \text{ we can apply } g\text{'s specification for } \alpha - 1 \text{]} \\ & \quad \text{fr + ex} \left| \begin{array}{l} (\text{list}_{\mathbb{N}}(x, v : vs') \star x = x \star r = \text{null} \star \alpha - 1 = 3|v : vs'| + 1 - (v \bmod 2)) \\ r := g(x); \\ (\text{list}_{\mathbb{N}}(x, v : vs') \star x = x \star r = |v : vs'| \star \alpha - 1 = 3|v : vs'| + 1 - (v \bmod 2)) \end{array} \right. \\ & \quad (\exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star x = x \star v = v \star \text{odd}(v) \star r = |v : vs| \star \alpha = 3|v : vs'| + (v \bmod 2)) \\ & \quad \left. \left(\exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star v = v \star r = |v : vs'| \star \right. \right. \\ & \quad \left. \left. (x = x' \star \text{even}(v) \vee x = x \star \text{odd}(v)) \star \alpha = 3|v : vs'| + (v \bmod 2) \right) \right) \\ & \quad \} \end{aligned}$$

To conclude the proof, we need to show that $Q'_{LL}(\alpha)$, $Q'_f(\alpha)$ and $Q'_g(\alpha)$ are in the internalisations of their external counterparts, which is done as follows, again eliding the proof for $Q_g(\alpha)$ as it is analogous to that of $Q_f(\alpha)$.

$$\begin{aligned} & \exists \vec{p}. Q'_{LL}(\alpha)[\vec{p}/\vec{p}] \star \text{ret} = r[\vec{p}/\vec{p}] \\ \Leftrightarrow & \exists p_x, p_r, p_v. ((x = \text{null} \star vs = \epsilon \star p_x = x \star p_r = |vs| \star p_v = \text{null} \star \alpha = 3|vs| + 2) \vee \\ & (\exists v, vs'. \text{list}_{\mathbb{N}}(x, v : vs') \star vs = v : vs' \star p_x = x \star p_v = v \star p_r = |vs| \\ & \quad \star \alpha = 3|vs| + 2)) \star \text{ret} = p_r \\ \Leftrightarrow & \text{list}_{\mathbb{N}}(x, vs) \star \text{ret} = |vs| \star \alpha = 3|vs| + 2 \\ & \exists \vec{p}. Q'_f(\alpha)[\vec{p}/\vec{p}] \star \text{ret} = r[\vec{p}/\vec{p}] \\ \Leftrightarrow & \exists p_x, p_v, p_r, x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star p_v = v \star p_r = |v : vs'| \\ & \quad \star (p_x = x' \star \text{even}(v) \vee p_x = x \star \text{odd}(v)) \star \alpha = 3|v : vs'| + (v \bmod 2) \\ \Leftrightarrow & \exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star \text{ret} = |v : vs'| \\ & \quad \star (\text{even}(v) \star \text{odd}(v)) \star \alpha = 3|v : vs'| + (v \bmod 2) \\ \Leftrightarrow & \exists x'. x \mapsto v, x' \star \text{list}_{\mathbb{N}}(x', vs') \star \text{ret} = |v : vs'| \star v \in \mathbb{N} \star \alpha = 3|v : vs'| + (v \bmod 2) \\ \Leftrightarrow & \text{list}_{\mathbb{N}}(x, v : vs') \star \text{ret} = |v : vs'| \star \alpha = 3|v : vs'| + (v \bmod 2) \end{aligned}$$

G COMPOSITIONAL SYMBOLIC EXECUTION

In this appendix, we present an outline of a proof of backward completeness for the below symbolic execution semantics.

G.1 Symbolic Execution Semantics

SKIP $\sigma, \text{skip} \Downarrow_{\Gamma} \text{ok} : \sigma$	ASSIGN $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{s}' = \hat{s}[x \mapsto \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), x := E \Downarrow_{\Gamma} \text{ok} : (\hat{s}', \hat{h}, \hat{\pi}')}$	ASSIGN (ERROR) $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\downarrow \hat{\pi}' \quad \hat{v}_{\text{err}} = [\text{"ExprEval"}, \text{str}(E)]}{(\hat{s}, \hat{h}, \hat{\pi}), x := E \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}')}$
NONDET $\frac{\hat{r} \text{ fresh} \quad \hat{\pi}' = \hat{r} \in \mathbb{N} \wedge \hat{\pi}}{(\hat{s}, \hat{h}, \hat{\pi}), x := \text{nondet} \Downarrow_{\Gamma} \text{ok} : (\hat{s}[x \mapsto \hat{r}], \hat{h}, \hat{\pi}')}$	ERROR $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\downarrow \hat{\pi}' \quad \hat{v}_{\text{err}} = [\text{"Error"}, \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{error}(E) \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}')}$	
ERROR (ERROR) $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\downarrow \hat{\pi}' \quad \hat{v}_{\text{err}} = [\text{"ExprEval"}, \text{str}(E)]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{error}(E) \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}')}$	IF-THEN $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{\pi}' \wedge \hat{v} \quad \text{SAT}(\hat{\pi}'') \quad (\hat{s}, \hat{h}, \hat{\pi}''), C_1 \Downarrow_{\Gamma} o : (\hat{s}', \hat{h}', \hat{\pi}''')}{(\hat{s}, \hat{h}, \hat{\pi}), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_{\Gamma} o : (\hat{s}', \hat{h}', \hat{\pi}''')}$	
IF-ELSE $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{\pi}' \wedge \neg \hat{v} \quad \text{SAT}(\hat{\pi}'') \quad (\hat{s}, \hat{h}, \hat{\pi}''), C_2 \Downarrow_{\Gamma} o : (\hat{s}', \hat{h}', \hat{\pi}''')}{(\hat{s}, \hat{h}, \hat{\pi}), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_{\Gamma} o : (\hat{s}', \hat{h}', \hat{\pi}''')}$	IF-ERR-VAL $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\downarrow \hat{\pi}' \quad \hat{v}_{\text{err}} = [\text{"ExprEval"}, \text{str}(E)]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}')}$	
IF-ERR-TYPE $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{\pi}' \wedge \hat{v} \notin \text{Bool} \quad \text{SAT}(\hat{\pi}'') \quad \hat{v}_{\text{err}} = [\text{"Type"}, \text{str}(E), \hat{v}, \text{"Bool"}]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{if } (E) C_1 \text{ else } C_2 \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}')}$	SEQ $\frac{\hat{\sigma}, C_1 \Downarrow_{\Gamma} \text{ok} : \hat{\sigma}' \quad \hat{\sigma}', C_2 \Downarrow_{\Gamma} o : \hat{\sigma}''}{\hat{\sigma}, C_1; C_2 \Downarrow_{\Gamma} o : \hat{\sigma}''}$ SEQ-ERR $\frac{\hat{\sigma}, C_1 \Downarrow_{\Gamma} o : \hat{\sigma}' \quad o \neq \text{ok}}{\hat{\sigma}, C_1; C_2 \Downarrow_{\Gamma} o : \hat{\sigma}'}$	
LOOKUP $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \hat{v}_m \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'')}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} \text{ok} : (\hat{s}[x \mapsto \hat{v}_m], \hat{h}, \hat{\pi}'')}$	LOOKUP-ERR-VAL $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\downarrow \hat{\pi}' \quad \hat{v}_{\text{err}} = [\text{"ExprEval"}, \text{str}(E)]}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}')}$	
LOOKUP-ERR-TYPE $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{v} \notin \mathbb{N} \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \hat{v}_{\text{err}} = [\text{"Type"}, \text{str}(E), \hat{v}, \text{"Nat"}]}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}'')}$		
LOOKUP-ERR-USE-AFTER-FREE $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \emptyset \quad \hat{\pi}'' = \hat{v} \in \mathbb{N} \wedge (\hat{v}_l = \hat{v}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \hat{v}_{\text{err}} = [\text{"UseAfterFree"}, \text{str}(E), \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}'')}$		
LOOKUP-ERR-MISSING $\frac{\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{v} \in \mathbb{N} \wedge \hat{v} \notin \text{dom}(\hat{h}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \hat{v}_{\text{err}} = [\text{"MissingCell"}, \text{str}(E), \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} \text{miss} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}'')}$	MUTATE $\frac{\llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}_1^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \hat{v}_m \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}_1) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \llbracket E_2 \rrbracket_{\hat{s}}^{\hat{\pi}''} \Downarrow \hat{v}_2^{\hat{\pi}'''} \quad \hat{h}' = \hat{h}[\hat{v}_l \mapsto \hat{v}_2]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{ok} : (\hat{s}, \hat{h}', \hat{\pi}''')}$	
MUTATE-ERR-VAL-1 $\frac{\llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\downarrow \hat{\pi}' \quad \hat{v}_{\text{err}} = [\text{"ExprEval"}, \text{str}(E_1)]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}')}$	MUTATE-ERR-TYPE $\frac{\llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}_1^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{v}_1 \notin \mathbb{N} \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \quad \hat{v}_{\text{err}} = [\text{"Type"}, \text{str}(E_1), \hat{v}_1, \text{"Nat"}]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{err} : (\hat{s}_{\text{err}}, \hat{h}, \hat{\pi}'')}$	

$$\begin{array}{c}
\text{MUTATE-ERR-MISSING} \\
\frac{[[E_1]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}_1^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{\sigma}_1 \in \mathbb{N} \wedge \hat{\sigma}_1 \notin \text{dom}(\hat{h}) \wedge \hat{\pi}' \\
\text{SAT}(\hat{\pi}'') \quad \hat{\sigma}_{err} = [\text{"MissingCell"}, \text{str}(E_1), \hat{\sigma}_1]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{miss} : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{MUTATE-ERR-USE-AFTER-FREE} \\
\frac{[[E_1]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}_1^{\hat{\pi}'} \quad \hat{h}(\hat{\sigma}_1) = \emptyset \quad \hat{\pi}'' = (\hat{\sigma}_1 = \hat{\sigma}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \\
\hat{\sigma}_{err} = [\text{"UseAfterFree"}, \text{str}(E_1), \hat{\sigma}]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{err} : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{MUTATE-ERR-VAL-2} \\
\frac{[[E_1]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}_1^{\hat{\pi}'} \quad [[E_2]]_s^{\hat{\pi}'} \Downarrow \hat{\sigma}_2^{\hat{\pi}''} \quad \hat{\pi}''' = \hat{\sigma}_1 \in \mathbb{N} \wedge \hat{\pi}'' \\
\text{SAT}(\hat{\pi}''') \quad \hat{\sigma}_{err} = [\text{"ExprEval"}, \text{str}(E_2)]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{err} : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{NEW} \\
\frac{\hat{l} \text{ fresh} \quad \hat{\pi}' = \hat{l} \in \mathbb{N} \wedge \hat{l} \notin \text{dom}(\hat{h}) \wedge \hat{\pi}}{(\hat{s}, \hat{h}, \hat{\pi}), x := \text{new}() \Downarrow_{\Gamma} \text{ok} : (\hat{s}[x \mapsto \hat{l}], \hat{h}[\hat{l} \mapsto \text{null}], \hat{\pi}')} \\
\\
\text{FREE} \\
\frac{[[E]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}^{\hat{\pi}'} \quad \hat{h}(\hat{\sigma}) = \hat{\sigma}_m \\
\hat{\pi}'' = (\hat{\sigma} = \hat{\sigma}) \wedge \hat{\pi}' \\
\text{SAT}(\hat{\pi}'') \quad \hat{h}' = \hat{h}[\hat{\sigma} \mapsto \emptyset]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} \text{ok} : (\hat{s}, \hat{h}', \hat{\pi}'')} \\
\\
\text{FREE-ERR-EVAL} \\
\frac{[[E]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}^{\hat{\pi}'} \quad \hat{\sigma}_{err} = [\text{"ExprEval"}, \text{str}(E)]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} \text{err} : (\hat{s}_{err}, \hat{h}, \hat{\pi}')} \\
\\
\text{FREE-ERR-TYPE} \\
\frac{[[E]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{\sigma} \notin \mathbb{N} \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \\
\hat{\sigma}_{err} = [\text{"Type"}, \text{str}(E), \hat{\sigma}, \text{"Nat"}]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} \text{err} : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{FREE-ERR-MISSING} \\
\frac{[[E]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}^{\hat{\pi}'} \quad \hat{\pi}'' = \hat{\sigma} \in \mathbb{N} \wedge \hat{\sigma} \notin \text{dom}(\hat{h}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \\
\hat{\sigma}_{err} = [\text{"MissingCell"}, \text{str}(E), \hat{\sigma}]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} \text{miss} : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{FREE-ERR-USE-AFTER-FREE} \\
\frac{[[E]]_s^{\hat{\pi}} \Downarrow \hat{\sigma}^{\hat{\pi}'} \quad \hat{h}(\hat{\sigma}) = \emptyset \quad \hat{\pi}'' = \hat{\sigma} \in \mathbb{N} \wedge (\hat{\sigma} = \hat{\sigma}) \wedge \hat{\pi}' \\
\text{SAT}(\hat{\pi}'') \quad \hat{\sigma}_{err} = [\text{"UseAfterFree"}, \text{str}(E), \hat{\sigma}]}{(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} \text{err} : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')} \\
\\
\text{FCALL} \\
\frac{[[\vec{E}]]_s^{\hat{\pi}} \Downarrow \vec{\sigma}^{\hat{\pi}'} \quad (\vec{x} = \vec{x} * P) \quad f(\vec{x}) \quad (ok : Q_{ok}) \quad (err : Q_{err}) \in \Gamma \quad \hat{\sigma} = [\vec{x} \mapsto \vec{\sigma}] \\
\text{matchAndConsume}(P, \hat{\sigma}, (\hat{s}, \hat{h}, \hat{\pi}')) \rightsquigarrow (\hat{\sigma}', \hat{h}_p, (\hat{s}, \hat{h}_f, \hat{\pi}''))^{ok} \quad r, \hat{r} \text{ fresh} \\
\text{produce}(Q_{ok}[r/\text{ret}], \hat{\sigma}'[r \mapsto \hat{r}], (\hat{s}, \hat{h}_f, \hat{\pi}'')) \rightsquigarrow (\hat{\sigma}'', \hat{h}_q, (\hat{s}, \hat{h}', \hat{\pi}'''))^{ok}}{(\hat{s}, \hat{h}, \hat{\pi}), y := f(\vec{E}) \Downarrow_{\Gamma} ok : (\hat{s}[y \mapsto \hat{r}], \hat{h}', \hat{\pi}''')} \\
\\
\text{FCALL-QERR} \\
\frac{[[\vec{E}]]_s^{\hat{\pi}} \Downarrow \vec{\sigma}^{\hat{\pi}'} \quad (\vec{x} = \vec{x} * P) \quad f(\vec{x}) \quad (ok : Q_{ok}) \quad (err : Q_{err}) \in \Gamma \quad \hat{\sigma} = [\vec{x} \mapsto \vec{\sigma}] \\
\text{matchAndConsume}(P, \hat{\sigma}, (\hat{s}, \hat{h}, \hat{\pi}')) \rightsquigarrow (\hat{\sigma}', \hat{h}_p, (\hat{s}, \hat{h}_f, \hat{\pi}''))^{ok} \quad r, \hat{r} \text{ fresh} \\
\text{produce}(Q_{err}[r/\text{err}], \hat{\sigma}'[r \mapsto \hat{r}], (\hat{s}, \hat{h}_f, \hat{\pi}'')) \rightsquigarrow (\hat{\sigma}'', \hat{h}_q, (\hat{s}, \hat{h}', \hat{\pi}'''))^{ok}}{(\hat{s}, \hat{h}, \hat{\pi}), y := f(\vec{E}) \Downarrow_{\Gamma} err : (\hat{s}[\text{err} \mapsto \hat{r}], \hat{h}', \hat{\pi}''')} \\
\\
\text{FCALL-ERR-VAL} \\
\frac{1 \leq m \leq n \quad \hat{\pi}_0 = \hat{\pi} \quad ([[E_i]]_s^{\hat{\pi}_{i-1}} \Downarrow \hat{\sigma}_i^{\hat{\pi}_i})_{i=1}^{m-1} \quad [[E_m]]_s^{\hat{\pi}_{m-1}} \Downarrow \hat{\sigma}_m^{\hat{\pi}'} \quad \hat{\sigma}_{err} = [\text{"ExprEval"}, \text{str}(E_m)]}{(\hat{s}, \hat{h}, \hat{\pi}), y := f(E_1, \dots, E_n) \Downarrow_{\Gamma} \text{err} : (\hat{s}_{err}, \hat{h}, \hat{\pi}')}
\end{array}$$

where $\hat{s}_{err} \triangleq \hat{s}[\text{err} \rightarrow \hat{\sigma}_{err}]$.

G.2 Backward Completeness

In this section, we show that the above symbolic execution semantics is backward complete w.r.t. the simple programming language used in ESL. This means, we prove the following theorem:

THEOREM G.1 (BACKWARD COMPLETENESS: SYMBOLIC EXECUTION).

$$\hat{\sigma}, C \Downarrow_{\Gamma} o : \hat{\sigma}' \wedge \models (\gamma, \Gamma) \implies \forall \sigma' \in \text{Mod}(\hat{\sigma}'). \exists \sigma \in \text{Mod}(\hat{\sigma}). \sigma, C \Downarrow_{\gamma} o : \sigma'$$

To relate smyboic expression evaluation to concrete evaluation, we require the following property.

PROPERTY 1.

$$\mathcal{W}f_{\hat{\pi}}(\hat{s}) \wedge \llbracket \tilde{E} \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{w}^{\hat{\pi}'} \wedge \varepsilon(\hat{\pi}') = \text{true} \wedge \hat{s} \subseteq \varepsilon \implies \llbracket \tilde{E} \rrbracket_{\varepsilon(\hat{s})} = \varepsilon(\hat{w})$$

where $\hat{w} \in \text{SVal} \cup \{\frac{1}{2}\}$.

Furthermore, we will require a lemma which says that any state that is well-formed w.r.t. a path condition is also well-formed with respect to a weaker path condition:

LEMMA G.2 (WF IMPLICATION).

$$\mathcal{W}f_{\hat{\pi}}(\hat{s}) \wedge (\hat{\pi}' \implies \hat{\pi}) \implies \mathcal{W}f_{\hat{\pi}'}(\hat{s})$$

PROOF. This is a straight-forward implication of the definition of well-formedness given in 6. \square

With this, we can now prove Theorem G.1.

PROOF. We assume

$$\hat{\sigma}, C \Downarrow_{\Gamma} o : \hat{\sigma}' \wedge \models (\gamma, \Gamma)$$

and prove by induction over the structure of C that

$$\forall \sigma' \in \text{Mod}(\hat{\sigma}'). \exists \sigma \in \text{Mod}(\hat{\sigma}). \sigma, C \Downarrow_{\gamma} o : \sigma'$$

All cases except function calls are straightforward. The successful function call and non-successful function call cases are similar, we present only the successful function call case here. We include a few simple representative cases for illustrative purposes.

Fcall. Rule:

$$\frac{\text{FCALL} \quad \llbracket \tilde{E} \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{\sigma}^{\hat{\pi}'} \quad (\vec{x} = \vec{x} * P) f(\vec{x}) (ok : Q_{ok}) (err : Q_{err}) \in \Gamma \quad \hat{\theta} = [\vec{x} \mapsto \vec{\sigma}] \quad \text{matchAndConsume}(P, \hat{\theta}, (\hat{s}, \hat{h}, \hat{\pi}')) \rightsquigarrow (\hat{\theta}', \hat{h}_p, (\hat{s}, \hat{h}_f, \hat{\pi}''))^{ok} \quad r, \hat{r} \text{ fresh} \quad \text{produce}(Q_{ok}[r/\text{ret}], \hat{\theta}'[r \mapsto \hat{r}], (\hat{s}, \hat{h}_f, \hat{\pi}'')) \rightsquigarrow (\hat{\theta}'', \hat{h}_q, (\hat{s}, \hat{h}', \hat{\pi}'''))^{ok}}{(\hat{s}, \hat{h}, \hat{\pi}), \gamma := f(\tilde{E}) \Downarrow_{\Gamma} ok : (\hat{s}[\gamma \mapsto \hat{r}], \hat{h}', \hat{\pi}''')}$$

We assume

$$(\hat{s}, \hat{h}, \hat{\pi}), \gamma := f(\tilde{E}) \Downarrow_{\Gamma} ok : (\hat{s}[\gamma \mapsto \hat{r}], \hat{h}', \hat{\pi}''')$$

which yields

$$(H1) \quad \llbracket \tilde{E} \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{\sigma}^{\hat{\pi}'}$$

$$(H2) \quad (\vec{x} = \vec{x} * P) f(\vec{x}) (ok : Q_{ok}) (err : Q_{err}) \in \Gamma$$

$$(H3) \quad \hat{\theta} = [\vec{x} \mapsto \vec{\sigma}]$$

$$(H4) \quad \text{matchAndConsume}(P, \hat{\theta}, (\hat{s}, \hat{h}, \hat{\pi}')) \rightsquigarrow (\hat{\theta}', \hat{h}_p, (\hat{s}, \hat{h}_f, \hat{\pi}''))^{ok}$$

$$(H4a) \quad \mathcal{W}f((\hat{s}, \hat{h}_f, \hat{\pi}''))$$

$$(H4b) \quad \hat{h} = \hat{h}_p \uplus \hat{h}_f$$

$$(H4c) \quad \hat{\pi}'' \Rightarrow \hat{\pi}'$$

$$(H4d) \quad \hat{\theta}' \geq \hat{\theta} \text{ and } \text{dom}(\hat{\theta}') = \{\vec{x}\} \cup \text{fv}(P)$$

$$(H4e) \quad (\emptyset, \hat{h}_p, \hat{\pi}'') \supseteq_{\mathcal{M}} P\hat{\theta}' \star \hat{\pi}'', \text{ i.e. } \forall \varepsilon, h_p. \varepsilon, \emptyset, h_p \models P\hat{\theta}' \star \hat{\pi}'' \implies \varepsilon((\emptyset, \hat{h}_p, \hat{\pi}'')) = (\emptyset, h_p)$$

$$(H5) \quad r, \hat{r} \text{ fresh}$$

$$(H6) \quad \text{produce}(Q_{ok}[r/\text{ret}], \hat{\theta}'[r \mapsto \hat{r}], (\hat{s}, \hat{h}_f, \hat{\pi}'')) \rightsquigarrow (\hat{\theta}'', \hat{h}_q, (\hat{s}, \hat{h}', \hat{\pi}'''))^{ok}$$

$$(H6a) \quad \mathcal{W}f((\hat{s}, \hat{h}', \hat{\pi}'''))$$

$$(H6b) \quad \hat{h}' = \hat{h}_q \uplus \hat{h}_f$$

$$(H6c) \quad \hat{\pi}''' \Rightarrow \hat{\pi}''$$

$$(H6d) \quad \hat{\theta}'' \geq \hat{\theta}'[r \mapsto \hat{r}] \text{ and } \text{dom}(\hat{\theta}'') = \{\vec{x}\} \cup \text{fv}(P) \cup \text{fv}(Q_{ok}[r/\text{ret}])$$

$$(H6e) \quad \forall \varepsilon. \varepsilon(\hat{\pi}''') = \text{true} \wedge \hat{\theta}'', \hat{s}, \hat{h}' \subseteq \varepsilon \implies \varepsilon(\hat{\theta}''), \emptyset, \varepsilon(\hat{h}_q) \models Q_{ok}[r/\text{ret}]$$

Now, let $\sigma' = (s', h') \in \text{Mod}(\hat{s}[\gamma \mapsto \hat{r}], \hat{h}', \hat{\pi}''')$, i.e. there exists an ε' such that

$$(H8a) \quad \varepsilon'(\hat{s}[\gamma \mapsto \hat{r}]) = s'$$

$$(H8b) \quad \varepsilon'(\hat{h}') = h'$$

(H8c) $\varepsilon'(\hat{\pi}''') = \text{true}$

(H8b), (H6b) imply that **(H9)** $\varepsilon'(\hat{h}') = \varepsilon'(\hat{h}_q) \uplus \varepsilon'(\hat{h}_f)$, and we define **(H9a)** $h_q = \varepsilon'(\hat{h}_q)$ and **(H9b)** $h_f = \varepsilon'(\hat{h}_f)$. With (H8b), this yields **(H9c)** $h' = h_q \uplus h_f$.

(H6e), (H8c), (H9a), together with an appropriately (arbitrarily) extended $\varepsilon'' \geq \varepsilon'$ that covers $\hat{\theta}''$ and \hat{s} imply **(H10)** $\varepsilon''(\hat{\theta}'')$, $\emptyset, h_q \models Q_{ok}[r/\text{ret}]$, and we define **(H10a)** $\theta = \varepsilon''(\hat{\theta}'')$.

(H2) and $\models (\gamma, \Gamma)$ imply that we have a valid UX triple¹¹ **(H11)** $\gamma \models [\vec{x} = \vec{x} * P * \vec{z} = \text{null}] \text{ C } [ok : Q'_{ok}]$, where **(H11a)** $Q_{ok} \Leftrightarrow \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \text{ret} = E'[\vec{p}/\vec{p}]$, **(H11b)** $f(\vec{x})\{C, \text{return } E'\} \in \gamma$ and **(H11c)** $\text{pv}(C) \setminus \{\vec{x}\} = \{\vec{z}\}$.

(H5), (H10), (H10a) and (H11a) imply

$$\theta, \emptyset, h_q \models \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star r = E'[\vec{p}/\vec{p}]$$

that is, that there exist $\vec{w} \in \text{Val}$, such that

$$\textbf{(H12)} \quad \theta, \emptyset[\vec{p} \mapsto \vec{w}], h_q \models Q'_{ok} \star r = E'$$

and we define **(H12a)** $s_q = \emptyset[\vec{p} \mapsto \vec{w}]$. Given (H10a), (H12) and (H6d), we obtain **(H13)** $\llbracket E' \rrbracket_{\theta, s_q} = \theta(r) = \varepsilon''(\hat{r})$.

(H9c), (H11) and (H12) imply

$$\textbf{(H14)} \quad \exists \tilde{s}_p, \tilde{h}_p. \theta, \tilde{s}_p, \tilde{h}_p \models \vec{x} = \vec{x} \star P \star \vec{z} = \text{null} \wedge (\tilde{s}_p, \tilde{h}_p \uplus h_f), C \Downarrow_{\gamma} ok : (s_q, h_q \uplus h_f)$$

(H3), (H4d), (H6d) and (H10a) then imply

$$\textbf{(H15)} \quad \theta(\vec{x}) = \varepsilon''(\hat{\theta}'(\vec{x})) = \varepsilon''(\vec{v})$$

Since P does not hold any program variables, (H14) implies

$$\textbf{(H16)} \quad \theta, \emptyset, \tilde{h}_p \models P$$

Given ε'' and (H4b), we can pick an $\varepsilon''' \geq \varepsilon''$ which covers \hat{h}_p , and obtain

$$\textbf{(H17)} \quad \varepsilon'''(\hat{h}) = \varepsilon'''(\hat{h}_p) \uplus \varepsilon'''(\hat{h}_f)$$

Lemma 6.3 (2), (H16) and (H10a) then imply

$$\textbf{(H18)} \quad \varepsilon''', \emptyset, \tilde{h}_p \models P\hat{\theta}''$$

(H8c), (H6c) and $\varepsilon''' \geq \varepsilon' \geq \varepsilon'$ also imply that

$$\textbf{(H19)} \quad \varepsilon'''(\hat{\pi}'') = \text{true}$$

which then, together with (H18), yields

$$\textbf{(H20)} \quad \varepsilon''', \emptyset, \tilde{h}_p \models P\hat{\theta}'' \star \hat{\pi}''$$

(H4d) and (H6d) yield $P\hat{\theta}'' = P\hat{\theta}'$, which together with (H4e) and (H20) yield

$$\textbf{(H21)} \quad \varepsilon'''(\hat{h}_p) = \tilde{h}_p,$$

meaning that we have matched the heaps. As C does not include program variables beyond \vec{x} and \vec{z} , (H14) and (H15) mean that we can restrict \tilde{s}_p to $\emptyset[\vec{x} \rightarrow \varepsilon''(\vec{v})][\vec{z} \rightarrow \text{null}]$ without affecting the execution of C , yielding **(H22)** $(\emptyset[\vec{x} \rightarrow \varepsilon''(\vec{v})][\vec{z} \rightarrow \text{null}], \tilde{h}_p \uplus h_f), C \Downarrow_{\gamma} : (s_q, h_q \uplus h_f)$. Finally, (H1) and Property 1 give that **(H23)** $\llbracket \vec{E} \rrbracket_s = \varepsilon'''(\vec{v})$, where **(H23a)** $s = \varepsilon'''(\hat{s})$.

The hypotheses needed to apply the concrete function call rule (for successful execution) are given through the hypotheses (H8a), (H11b), (H11c), (H13), (H22) and (H23). This yields

$$\textbf{(H23)} \quad (s, \tilde{h}_p \uplus h_f), x := f(\vec{E}) \Downarrow_{\gamma} (s', h_q \uplus h_f)$$

(H9b), (H17), (H21) and $\varepsilon''' \geq \varepsilon'$ yield **(H25)** $\tilde{h}_p \uplus h_f = \varepsilon'''(\hat{h})$.

Lastly, (H1) implies $\hat{\pi}'' \Rightarrow \hat{\pi}$, which together with (H4c), (H6c) and (H8c) yields **(H26)** $\varepsilon'''(\hat{\pi}) = \text{true}$.

¹¹A UX triple is valid when frame-preserving under-approximating validity as defined in Def. 4.5 holds

Finally, (H23a), (H25) and (H26) imply

$$(s, \tilde{h}_p \uplus h_f) \in \text{Mod}((\hat{s}, \hat{h}, \hat{\pi}))$$

which concludes the proof.

Assign. Rule:

$$\frac{\text{ASSIGN} \quad \llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{s}' = \hat{s}[x \mapsto \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), x := E \Downarrow_{\Gamma} \text{ok} : (\hat{s}', \hat{h}, \hat{\pi}')} \quad \text{We assume}$$

We assume

$$(\hat{s}, \hat{h}, \hat{\pi}), x := E \Downarrow_{\Gamma} \text{ok} : (\hat{s}[x \mapsto \hat{v}], \hat{h}, \hat{\pi}')$$

which yields

$$\text{(H0)} \quad \mathcal{W}f_{\hat{\pi}}(\hat{s})$$

$$\text{(H1)} \quad \llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'}$$

$$\text{(H2)} \quad \hat{s}' = \hat{s}[x \mapsto \hat{v}]$$

Now, let $\sigma' = (s', h') \in \text{Mod}(\hat{s}[x \mapsto \hat{v}], \hat{h}, \hat{\pi}')$, i.e. there exists an ε' such that

$$\text{(H3a)} \quad \varepsilon'(\hat{s}[x \mapsto \hat{v}]) = s'$$

$$\text{(H3b)} \quad \varepsilon'(\hat{h}) = h'$$

$$\text{(H3c)} \quad \varepsilon'(\hat{\pi}') = \text{true}$$

(H1) implies **(H4)** $\hat{\pi}' \Rightarrow \hat{\pi}$.

Furthermore, let $\varepsilon \geq \varepsilon'$ be an extension which covers \hat{s} and define **(H5)** $s = \varepsilon(\hat{s})$. (H4) and (H3c) imply **(H6)** $\varepsilon(\hat{\pi}') = \text{true}$. Given (H0), (H1), (H5) and (H6), Property 1 imply **(H7)** $\varepsilon(\hat{v}) = \llbracket E \rrbracket_s$

Defining $v = \llbracket E \rrbracket_s$, (H3a), (H5) and (H7) yield **(H8)** $s' = s[x \mapsto v]$.

The concrete operational semantics therefore yields

$$(s, h'), x := E \Downarrow_{\gamma} (s', h')$$

and through (H5), (H3b) and (H6), we obtain $(s, h') \in \text{Mod}((\hat{s}, \hat{h}, \pi))$, which concludes the proof.

Mutate. Rule:

$$\frac{\begin{array}{l} \text{MUTATE} \\ \llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}_1^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \hat{v}_m \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}_1) \wedge \hat{\pi}' \\ \text{SAT}(\hat{\pi}'') \quad \llbracket E_2 \rrbracket_{\hat{s}}^{\hat{\pi}''} \Downarrow \hat{v}_2^{\hat{\pi}'''} \quad \hat{h}' = \hat{h}[\hat{v}_l \mapsto \hat{v}_2] \end{array}}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{ok} : (\hat{s}, \hat{h}', \hat{\pi}''')} \quad \text{We assume}$$

We assume

$$(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} \text{ok} : (\hat{s}, \hat{h}', \hat{\pi}''')$$

which yields

$$\text{(H0)} \quad \mathcal{W}f_{\hat{\pi}}(\hat{s}) \text{ and } \mathcal{W}f_{\hat{\pi}}(\hat{h})$$

$$\text{(H1)} \quad \llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}_1^{\hat{\pi}'}$$

$$\text{(H2)} \quad \hat{h}(\hat{v}_l) = \hat{v}_m$$

$$\text{(H3)} \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}_1) \wedge \hat{\pi}'$$

$$\text{(H4)} \quad \text{SAT}(\hat{\pi}'')$$

$$\text{(H5)} \quad \llbracket E_2 \rrbracket_{\hat{s}}^{\hat{\pi}''} \Downarrow \hat{v}_2^{\hat{\pi}'''}$$

$$\text{(H6)} \quad \hat{h}' = \hat{h}[\hat{v}_l \mapsto \hat{v}_2]$$

Now, let $\sigma' = (s, h') \in \text{Mod}(\hat{s}, \hat{h}', \hat{\pi}''')$, i.e. (given (H6)) there exists an ε' such that

$$\text{(H7a)} \quad \varepsilon'(\hat{s}) = s$$

$$\text{(H7b)} \quad \varepsilon'(\hat{h}[\hat{v}_l \mapsto \hat{v}_2]) = h'$$

$$\text{(H7c)} \quad \varepsilon'(\hat{\pi}''') = \text{true}$$

(This ε' exists, because (H4) and (H5) imply $\text{SAT}(\hat{\pi}''')$.)

(H1), (H3) and (H5) imply **(H8)** $\hat{\pi}''' \Rightarrow \hat{\pi}'' \Rightarrow \hat{\pi}' \Rightarrow \hat{\pi}$ and (H7c) yields **(H9)** $\varepsilon'(\hat{\pi}''') = \varepsilon'(\hat{\pi}'') = \varepsilon'(\hat{\pi}') = \varepsilon'(\hat{\pi}) = \text{true}$.

(H0), (H2) and (H9) implies **(H10)** $\varepsilon'(\hat{v}_l) \in \mathbb{N}$.

(H3) and (H9) imply **(H11a)** $\varepsilon'(\hat{v}_l) = \varepsilon'(\hat{v}_1)$ and we define **(H11a)** $n = \varepsilon'(\hat{v}_l) = \varepsilon'(\hat{v}_1) \in \mathbb{N}$, given (H10).

We extend $\varepsilon \geq \varepsilon'$ to cover \hat{h} and define **(H12a)** $h = \varepsilon(\hat{h})$ and (H11a) and (H12) then implies **(H12b)** $h(n) \in \text{Val}$.

(H6), (H7b), (H11a) and (H12a) implies **(H13)** $h' = h[n \mapsto \varepsilon'(\hat{v}_2)]$.

Given (H0), (H1), (H7a), (H9) and (H11a), Property 1 implies **(H14)** $\llbracket E_1 \rrbracket_s = n$.

Given (H0), (H5) and (H8), G.2 implies **(H15)** $\mathcal{W}f_{\hat{\pi}''}(\hat{h})$.

Given (H5), (H7a), (H9) and (H15), Property 1 yields **(H16)** $\llbracket E_2 \rrbracket_s = \varepsilon'(\hat{v}_2)$.

Given (H12b), (H13), (H14) and (H16), the concrete semantics yields

$$(s, h), [E_1] := E_2 \Downarrow_Y (s, h')$$

and since (H7a), (H8) and (H13) implies $(\sigma, h) \in \text{Mod}((\hat{s}, \hat{h}, \hat{\pi}))$, concluding the proof.

Free. Rule:

$$\begin{array}{c} \text{FREE} \\ \frac{\begin{array}{c} \llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \hat{v}_m \\ \hat{\pi}'' = (\hat{v}_l = \hat{v}) \wedge \hat{\pi}' \\ \text{SAT}(\hat{\pi}'') \quad \hat{h}' = \hat{h}[\hat{v}_l \mapsto \emptyset] \end{array}}{(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} \text{ok} : (\hat{s}, \hat{h}', \hat{\pi}'')} \end{array}$$

We assume

$$(\hat{s}, \hat{h}, \hat{\pi}), \text{free}(E) \Downarrow_{\Gamma} \text{ok} : (\hat{s}, \hat{h}[\hat{v}_l \mapsto \emptyset], \hat{\pi}'')$$

which yields

(H0) $\mathcal{W}f_{\hat{\pi}}(\hat{s})$ and $\mathcal{W}f_{\hat{\pi}}(\hat{h})$

(H1) $\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'}$

(H2) $\hat{h}(\hat{v}_l) = \hat{v}_m$

(H3) $\hat{\pi}'' = (\hat{v}_l = \hat{v}) \wedge \hat{\pi}'$

(H4) $\text{SAT}(\hat{\pi}'')$

(H5) $\hat{h}' = \hat{h}[\hat{v}_l \mapsto \emptyset]$

Now, let $\sigma' = (s, h') \in \text{Mod}(\hat{s}, \hat{h}', \hat{\pi}'')$, i.e. (given (H5)) there exists an ε' such that

(H6a) $\varepsilon'(\hat{s}) = s$

(H6b) $\varepsilon'(\hat{h}[\hat{v}_l \mapsto \emptyset]) = h'$

(H6c) $\varepsilon'(\hat{\pi}'') = \text{true}$

(H1) and (H3) imply **(H7)** $\hat{\pi}'' \Rightarrow \hat{\pi}' \Rightarrow \hat{\pi}$ and through (H6c) we obtain **(H8)** $\varepsilon'(\hat{\pi}'') = \varepsilon'(\hat{\pi}') = \varepsilon'(\hat{\pi}) = \text{true}$.

Extending $\varepsilon \geq \varepsilon'$ to cover \hat{h} , we define **(H9)** $h = \varepsilon(\hat{h})$.

Given (H0), (H1), (H8) and (H6a), G.2 yields **(H10a)** $\llbracket E \rrbracket_s = \varepsilon'(\hat{v})$. Defining $n = \varepsilon'(\hat{v})$, (H0), (H2), (H3), (H8) and (H10a) imply **(H10b)** $n = \llbracket E \rrbracket_s = \varepsilon'(\hat{v}) = \varepsilon'(\hat{v}_l)$. With (H2) and (H9) we obtain **(H10c)** $h(n) \in \text{Val}$.

(H6b), (H9) and (H10b) yield **(H11)** $h' = h[n \mapsto \emptyset]$.

Given (H10b), (H10c) and (H11), the concrete semantics yields

$$(s, h), \text{free}(E) \Downarrow_Y (s, h')$$

and (H6a), (H8), (H9) and $\varepsilon \geq \varepsilon'$ imply $(s, h) \in \text{Mod}((\hat{s}, \hat{h}, \hat{\pi}))$, concluding the proof.

Seq. Rule:

$$\begin{array}{c} \text{SEQ} \\ \frac{\hat{\sigma}, C_1 \Downarrow_{\Gamma} \text{ok} : \hat{\sigma}' \quad \hat{\sigma}', C_2 \Downarrow_{\Gamma} \text{ok} : \hat{\sigma}''}{\hat{\sigma}, C_1; C_2 \Downarrow_{\Gamma} \text{ok} : \hat{\sigma}''} \end{array}$$

We assume

$$\hat{\sigma}, C_1; C_2 \Downarrow_{\Gamma} \text{ok} : \hat{\sigma}''$$

which yields

$$(H1) \quad \hat{\sigma}, C_1 \Downarrow_{\Gamma} ok : \hat{\sigma}'$$

$$(H2) \quad \hat{\sigma}', C_2 \Downarrow_{\Gamma} o : \hat{\sigma}''$$

Now, let $\sigma' \in \text{Mod}(\hat{\sigma})$, i.e. there exists some ε such that $\sigma = \varepsilon'(\hat{\sigma})$. The inductive hypothesis and (H2) imply that there exists some $\sigma'' \in \text{Mod}(\hat{\sigma}'')$ such that (H3) $\sigma'', C_2 \Downarrow_Y \varepsilon'(\hat{r}) : \sigma'$. The inductive hypothesis and (H1) imply that there exists some $\sigma \in \text{Mod}(\hat{\sigma})$ such that (H4) $\sigma, C_1 \Downarrow_Y \sigma''$.

Given (H3) and (H4), the concrete semantics imply

$$\sigma, C_1; C_2 \Downarrow_Y \varepsilon'(\hat{r}) : \sigma'$$

As $\sigma \in \text{Mod}(\hat{\sigma})$, the proof is concluded.

Mutate-Err-Val-1. Rule:

$$\frac{\text{MUTATE-ERR-VAL-1} \quad \llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\Downarrow \hat{\pi}' \quad \hat{v}_{err} = [\text{"ExprEval"}, \text{str}(E_1)]}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} err : (\hat{s}_{err}, \hat{h}, \hat{\pi}')}$$

We assume

$$(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} err : (\hat{s}_{err}, \hat{h}, \hat{\pi}')$$

which yields

$$(H1) \quad \mathcal{W}f_{\hat{\pi}}(\hat{s})$$

$$(H2) \quad \llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \not\Downarrow \hat{\pi}'$$

$$(H3) \quad \hat{v}_{err} = [\text{"ExprEval"}, \text{str}(E_1)]$$

Now, let $(s', h) \in \text{Mod}((\hat{s}_{err}, \hat{h}, \hat{\pi}'))$, i.e. there exists some ε' such that

$$(H4a) \quad \varepsilon'(\hat{s}[\text{err} \mapsto \hat{v}_{err}]) = s'$$

$$(H4b) \quad \varepsilon'(\hat{h}) = h$$

$$(H4c) \quad \varepsilon'(\hat{\pi}') = \text{true}$$

(H2) implies (H5) $\hat{\pi}' \Rightarrow \hat{\pi}$, which implies with (H4c) that (H6) $\varepsilon'(\hat{\pi}') = \varepsilon'(\hat{\pi}) = \text{true}$.

Define (H7) $s = \varepsilon'(\hat{s})$.

Given (H1), (H2), (H6) and (H7), Property 1 yields (H8) $\llbracket E_1 \rrbracket_s = \not\Downarrow$.

As \hat{v}_{err} has no symbolic variables, we have (H9) $v_{err} = \varepsilon'(\hat{v}_{err}) = \hat{v}_{err}$.

(H4a), (H7) and (H9) implies that (H10) $s' = s[\text{err} \mapsto v_{err}]$.

Given (H10), (H8), (H9) and (H3), the operational semantics implies

$$(s, h), [E_1] := E_2 \Downarrow_Y err : (s', h)$$

(H7), (H4b) and (H6) imply that $(s, h) \in \text{Mod}(\hat{s}, \hat{h}, \hat{\pi})$, concluding the proof.

Mutate-Err-Use-After-Free. Rule:

$$\frac{\text{MUTATE-ERR-USE-AFTER-FREE} \quad \begin{array}{l} \llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'} \quad \hat{h}(\hat{v}_l) = \emptyset \\ \hat{\pi}'' = (\hat{v}_l = \hat{v}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \\ \hat{v}_{err} = [\text{"UseAfterFree"}, \text{str}(E_1), \hat{v}] \end{array}}{(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} err : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')}$$

We assume

$$(\hat{s}, \hat{h}, \hat{\pi}), [E_1] := E_2 \Downarrow_{\Gamma} err : (\hat{s}_{err}, \hat{h}, \hat{\pi}'')$$

which yields

$$(H0) \quad \mathcal{W}f_{\hat{\pi}}(\hat{s})$$

$$(H1) \quad \llbracket E_1 \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow \hat{v}^{\hat{\pi}'}$$

$$(H2) \quad \hat{h}(\hat{v}_l) = \emptyset$$

$$(H3) \quad \hat{\pi}'' = (\hat{v}_l = \hat{v}) \wedge \hat{\pi}'$$

$$(H4) \quad \text{SAT}(\hat{\pi}'')$$

$$(H5) \quad \hat{v}_{err} = [\text{"UseAfterFree"}, \text{str}(E_1), \hat{v}]$$

Now, let $(s', h) \in \text{Mod}((\hat{s}_{err}, \hat{h}, \hat{\pi}''))$, i.e. there exists some ε' such that

(H6a) $\varepsilon'(\hat{s}[\text{err} \mapsto \hat{v}_{err}]) = s'$

(H6b) $\varepsilon'(\hat{h}) = h$

(H6c) $\varepsilon'(\hat{\pi}'') = \text{true}$

(H1) and (H3) imply **(H7)** $\hat{\pi}'' \Rightarrow \hat{\pi}' \Rightarrow \hat{\pi}$, which implies with (H6c) that **(H8)** $\varepsilon'(\hat{\pi}'') = \varepsilon'(\hat{\pi}') = \varepsilon'(\hat{\pi}) = \text{true}$ and **(H9)** $\varepsilon'(\hat{v}_l) = \varepsilon'(\hat{v})$.

Define **(H10)** $s = \varepsilon'(\hat{s})$.

Given (H0), (H1), (H8), (H10) and (H6a), Property 1 yields **(H11)** $\llbracket E_1 \rrbracket_s = \varepsilon'(\hat{v}) \in \mathbb{N}$.

(H2), (H3) and (H6b) imply **(H12)** $h(\varepsilon'(\hat{v}_l)) = \emptyset$.

(H5) implies **(H13)** $\varepsilon'(\hat{v}_{err}) = [\text{"UseAfterFree"}, \text{str}(E_1), \varepsilon'(\hat{v})]$.

(H6a), (H10) and (H13) imply **(H14)** $s' = s[\text{err} \mapsto \varepsilon'(\hat{v}_{err})]$.

Given (H9), (H11), (H12), (H13) and (H14), the concrete semantics imply

$$(s, h), [E_1] := E_2 \Downarrow_{\gamma} \text{err} : (s', h)$$

(H10), (H6b), and (H8) and imply that $(s, h) \in \text{Mod}((\hat{s}, \hat{h}, \hat{\pi}))$, concluding the proof. □