Exact Separation Logic

Towards Bridging the Gap Between Verification and Bug-Finding

Petar Maksimović
Imperial College London, UK
Runtime Verification Inc., Urbana, IL, USA

Caroline Cronjäger
Ruhr-Universität Bochum, Germany

Andreas Lööw
Imperial College London, UK

Julian Sutherland
Nethermind, London, UK

Philippa Gardner
Imperial College London, UK

Abstract

Over-approximating (OX) program logics, such as separation logic (SL), are used for *verifying* properties of heap-manipulating programs: all terminating behaviour is characterised, but established 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 Pulse-X. In contrast, under-approximating (UX) program logics, such as incorrectness separation logic, are used to *find* true results and bugs: established results and errors are reachable, but there is no mechanism for understanding if all terminating behaviour has been characterised.

We introduce exact separation logic (ESL), which provides fully-verified function specifications compatible with both OX verification and UX true bug-funding: all terminating behaviour is characterised and all established results and errors are reachable. We prove soundness for ESL with mutually recursive functions, demonstrating, for the first time, function compositionality for a UX logic. We show that UX program logics require subtle definitions of internal and external function specifications compared with the familiar definitions of OX logics. We investigate the expressivity of ESL and, for the first time, explore the role of abstraction in UX reasoning by verifying abstract ESL specifications of various data-structure algorithms. In doing so, we highlight the difference between *abstraction* (hiding information) and *over-approximation* (losing information). Our findings demonstrate that abstraction cannot be used as freely in UX logics as in OX logics, but also that it should be feasible to use ESL to provide tractable function specifications for self-contained, critical code, which would then be used for both verification and true bug-finding.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Program reasoning; Theory of computation → Separation logic; Theory of computation → Hoare logic; Theory of computation → Abstraction

Keywords and phrases Separation logic, program correctness, program incorrectness, abstraction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.19

Related Version *Extended Version*: `https://arxiv.org/abs/2208.07200`

Funding Maksimović, Lööw and Gardner were partially supported by the EPSRC Fellowship "VetSpec: Verified Trustworthy Software Specification" (EP/R034567/1). Cronjäger was partially supported by the Erasmus Plus Student Mobility for Traineeships scheme.

Acknowledgements We would like to thank Sacha-Élie Ayoun and Daniele Nantes Sobrinho for the many discussions that have improved the quality of the paper. We would also like to thank the anonymous reviewers for their comments.

1 Introduction

Over-approximating (OX) program logics were introduced to reason about program correctness, starting with Hoare logic [18] and evolving to separation logic (SL) [27, 30]. SL is used for *verification* and features function specifications of the form $\{P\}\ f(\vec{x})\ \{Q\}$, the meaning of which is that all terminating executions of the function f that start from a state in the pre-condition P end in a state covered by the post-condition Q. SL has the standard rule of *forward consequence*, which allows one to *lose information* (for example, if we had a post-condition with $x = 42$, we could soundly weaken this precise information to the less precise $x > 0$ or even to the non-informative `true`). In essence, the philosophy underlying the OX approach in general can be stated as:

no paths can be cut, but information can be lost.

A key property of SL is that function specifications are *compositional*, enabling *scalable* reasoning about the heap. This is due to their *locality*, which allows the pre-condition to describe only the partial state sufficient for the function to execute, and the *frame* property, which allows the function to be called in any larger state. SL function specifications have been used for verification of complex, real-world code in tools such as VeriFast [19], Iris [20], and Gillian [10, 23]. However, given that their post-conditions may describe states that are not reachable from their pre-conditions, such OX specifications are not compatible with true bug-finding, as found, for example, in Meta's Pulse [28] and Pulse-X [21] tools.

Under-approximating (UX) program logics were recently introduced, originating from reverse Hoare logic (RHL) [8] for reasoning about correctness of probabilistic programs, and coming to prominence with incorrectness logic [26] and incorrectness separation logic (ISL) [28], which identified their bug-finding potential. ISL function specifications are of the form $[P]\ f(\vec{x})\ [ok : Q_{ok}]$ and $[P]\ f(\vec{x})\ [err : Q_{err}]$, the meaning of which is that any state in the success post-condition Q_{ok} or the error post-condition Q_{err} is reachable from some state in the pre-condition P by executing the function f; this guarantees that all results and bugs reported in the post-conditions will be true. In contrast to SL, ISL uses the rule of *backward consequence*, which allows one to *cut paths* (for example, if we had a post-condition with $x > 0$, we could soundly strengthen this information to consider only the path in which $x = 42$). Therefore, the philosophy underlying UX logics in general can be summarised as:

paths can be cut, but no information can be lost.

When it comes to the use of ISL function specifications, whilst this has been implemented in Pulse-X, as far as we are aware, ISL does not feature function-call rules, and function compositionality for ISL and UX logics has not been proven. Moreover, as it is not possible to determine if UX specifications cover all terminating behaviour, they remain incompatible with verification and cannot therefore be used in tools such as VeriFast, Iris, and Gillian.

Our challenge is to develop a program logic in which we can state and prove function specifications that are compatible with *both* verification and true bug-finding. Our motivation comes from the unique flexibility and expressivity that such specifications would provide, as they could be used by verification and bug-finding tools alike, closing the gap between these two contrasting paradigms. From our experience in program logics and associated tool-building, we believe that the main use case for exact specification should be self-contained, critical code, such as widely-used data-structure libraries.

We introduce exact separation logic (ESL), with *exact* (EX) function specifications of the form $(P)\ f(\vec{x})\ (ok : Q_{ok})\ (err : Q_{err})$, whose meaning combines that of SL and ISL specifications: all terminating executions of the function that start from a state in the

pre-condition P end in a state covered by the post-conditions; *and* all states in two post-conditions are reachable from a state in the pre-condition by executing the function. The exactness of ESL can be captured by the slogan:

no paths can be cut and no information can be lost.

The slogan is supported by the rule of *equivalence*, which combines the forward consequence of SL and the backward consequence of ISL. In fact, ESL proof rules form a common core of SL and ISL, and ESL should therefore be a familiar setting to those acquainted with either.

We prove soundness for ESL with mutually recursive functions, which we believe is the first proof of function compositionality for a UX logic, and which transfers immediately to ISL. In doing so, drawing inspiration from InsecSL [25], we provide formal definitions of *external* and *internal* function specifications, which describe, respectively, the interface a function exposes towards its clients and towards its implementation, and highlight the difference in complexity between these two types of specifications in OX and UX reasoning.

Using numerous examples, we demonstrate here and in the extended version [24] how ESL can be used to reason about data-structure libraries, language errors, mutual recursion, and non-termination. In doing so, we introduce, for the first time, abstract predicates to UX reasoning and provide abstract function specifications for a number of data-structure algorithms, focussing on singly-linked lists and binary trees. In doing so, we highlight an important difference between the concepts of abstraction and over-approximation: in particular, abstraction corresponds to *hiding* information whereas over-approximation corresponds to *losing* it. Our findings demonstrate that, while abstraction cannot be used as freely in UX logics as in OX logics, sometimes resulting in less abstract specifications and more complex proofs, it should be feasible to use ESL to provide tractable function specifications for self-contained, critical code that can then be used for both verification and true bug-finding.

2 Exact Separation Logic by Example

We guide the reader through what it means to write ESL specifications and proofs by intuition and example, contrasting our findings with those known from SL and ISL.

Illustrative Example. Consider the command $C \triangleq \texttt{if}\ (\mathsf{x} > 0)\ \{\mathsf{y} := 42\}\ \texttt{else}\ \{\mathsf{y} := 21\}$, which can be specified, starting from the pre-condition $\mathsf{x} \in \mathbb{Z}$, in ESL, SL, and ISL as follows:

$(\ \mathsf{x} \in \mathbb{Z}\ )$
`if` $(\mathsf{x} > 0)$ {
  $(\ \mathsf{x} > 0\ )$
  $\mathsf{y} := 42$
  $(\ Q_1 : \mathsf{x} > 0 \land \mathsf{y} = 42\ )$
} `else` {
  $(\ \mathsf{x} \leq 0\ )$
  $\mathsf{y} := 21$
  $(\ Q_2 : \mathsf{x} \leq 0 \land \mathsf{y} = 21\ )$
}
$(\ Q_1 \lor Q_2\ )$

$\{\ \mathsf{x} \in \mathbb{Z}\ \}$
`if` $(\mathsf{x} > 0)$ {
  ...
  // Same as ESL
  ...
}
$\{\ Q_1 \lor Q_2\ \}$
// Losing information
$\{\ \mathsf{x} \in \mathbb{Z} \land \mathsf{y} > 0\ \}$

$[\ \mathsf{x} \in \mathbb{Z}\ ]$
`if` $(\mathsf{x} > 0)$ {
  $[\ \mathsf{x} > 0\ ]$
  $\mathsf{y} := 42$
  $[\ Q_1 : \mathsf{x} > 0 \land \mathsf{y} = 42\ ]$
} `else` { $\mathsf{y} := 21$ }
// Path cutting
$[\ \mathsf{x} > 0 \land \mathsf{y} = 42\ ]$

As ESL specifications must neither cut paths nor lose information (in this example, about the values of x and y), the ESL post-condition of C must be equivalent to $(\mathsf{x} > 0 \land \mathsf{y} = 42) \lor (\mathsf{x} \leq 0 \land \mathsf{y} = 21)$. In SL, it is possible to use forward consequence to weaken this

information and obtain, for example, $\mathsf{x} \in \mathbb{Z} \land \mathsf{y} > 0$, or just $\mathsf{x} \in \mathbb{Z}$, or even just true. In ISL, it is possible to cut, for example, the `else` branch of the `if` statement, but the values of x and y must be maintained in the post-condition of the `then` branch, $\mathsf{x} > 0 \land \mathsf{y} = 42$.

One question that we have been often asked is whether it is simpler to prove an exact specification $(P)\ \mathsf{C}\ (ok : Q_{ok})\ (err : Q_{err})$ in ESL, or to prove it separately in SL and ISL. The answer is that it is simpler to prove the specification in ESL. If a specification is exact, then it does not cut paths and it does not lose information. Therefore, the tools that make SL and ISL proofs simpler than ESL proofs, namely forward consequence and backward consequence, can only be used in very limited ways, if at all. From our experience, the ISL proof of an exact specification will turn out to be almost identical to the ESL one, and an SL proof on top of that would duplicate a large part of the work. In fact, if one were to try to prove the exact specification $(\mathsf{x} \in \mathbb{Z})\ C\ ((\mathsf{x} > 0 \land \mathsf{y} = 42) \lor (\mathsf{x} \leq 0 \land \mathsf{y} = 21))$ from the above example in either SL or ISL, they would obtain exactly the same proof as in ESL.

We also emphasise that ESL is not meant to replace either SL or ISL. If one is interested in only verification or only bug-finding, then one should use a formalism tailored to that type of analysis to exploit the available shortcuts. However, if one wanted to use the same codebase for both verification and bug-finding, then ESL offers a way of providing specifications useful for both. One example of such a codebase would be a widely-used data-structure library, where some of the users use it for verification and others for bug-finding.

List-length in ESL: Intuition. We consider a list-length function, $\mathtt{LLen}(\mathsf{x})$, which takes a list at x, does not modify it, and returns its length, and the following ESL specification:

$$(\mathsf{x} = x \star \mathsf{list}(x, n))\ \ \mathtt{LLen}(\mathsf{x})\ \ (ok : \mathsf{list}(x, n) \star \mathsf{ret} = n)$$

This specification uses a standard list-length predicate, $\mathsf{list}(x, n)$, which states that the length of the list at x equals n and is defined as follows:

$$\mathsf{list}(x, n) \triangleq (x = \mathtt{null} \star n = 0) \lor (\exists v, x'.\, x \mapsto v, x' \star \mathsf{list}(x', n-1)),$$

hiding the information about the values and internal node addresses of the list. Before proving this specification, we establish some intuition about why it holds. Let us assume that it does not hold and try to find a counter-example: by the meaning of ESL specifications, it is either not OX-valid or it is not UX-valid. The former, however, is not possible, as the analogous SL specification holds. The latter means that it is possible to find a state in the post-condition not reachable by the execution of f from any state in the pre-condition, and may be unfamiliar to the reader as UX program logics have been introduced only recently.

We start looking for such a state in the post-condition (post-model) by choosing some values for x and n: say, $x = 0$ and $n = 2$. This also fixes ret to 2. Then, we fully unfold $\mathsf{list}(0, 2)$ to obtain $\exists v_1, x_1, v_2.\ 0 \mapsto v_1, x_1 \star x_1 \mapsto v_2, \mathtt{null}$, and instantiate the existentials v_1, x_1, and v_2: say, with 1, 4, and 9, respectively. In this way, we obtain the state described by the assertion $0 \mapsto 1, 4 \star 4 \mapsto 9, \mathtt{null}$. When it comes to the pre-condition, x and n (and also x) are fixed by the post-model choices, and when we unfold the list, the pre-condition becomes $\mathsf{x} = 0 \star \exists v_1, x_1, v_2.\ 0 \mapsto v_1, x_1 \star x_1 \mapsto v_2, \mathtt{null}$. As the algorithm does not modify the list, it becomes clear that if we choose v_1, x_1, and v_2 as for the post-model (that is, 1, 4, and 9, respectively), the algorithm will reach our post-model. Given that the same reasoning would apply for any choice of x and n, we realise that the given specification is, in fact, also UX-valid and hence exact. This reveals an important observation, which is that

abstraction does not always equate to over-approximation, that is,
hiding information does not always mean losing information.

For those used to OX reasoning, it might appear that the post-condition $\mathsf{list}(x, n) \star \mathsf{ret} = n$ loses information about the structure of the list, but the insight here is that this information was never known in the pre-condition in the first place, as we also only had $\mathsf{list}(x, n)$ there.

List-length in ESL: Proof Sketch. Reasoning about function specifications in the UX/EX setting has not been studied previously and requires 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 are well-understood and the gap between them is small. For UX/EX logics, this gap is larger. We illustrate these concepts informally using the list-length example, and give the corresponding formal definitions in §4.

The proof sketch of the ESL external specification of the list-length algorithm is given in Figure 1. It is more complex than its SL counterpart (cf. [24]), but is manageable and comes with the benefit that this ESL specification can be used for both verification and bug-finding.

First, as the function is recursive, we have to provide a measure and prove the specification extended with this measure: in this case, the measure is $\alpha = n$, given by the length of the list. This measure is necessary to ensure the finite reachability property for mutually recursive functions in UX logics, and is a known technique from the work on total correctness specifications for OX logics [7, 9]. Recursive function calls are then allowed only if they use specifications of a strictly smaller measure, represented in the proof sketch by the function specification context $\Gamma(\alpha)$, which contains the specification of $\mathtt{LLen}(\mathsf{x})$ for all $\beta < \alpha$.

The move from the external to the internal pre-condition initialises the local function variables to `null`. The ESL rule for the `if` statement, just like in SL, adds the condition to the then-branch, its negation to the else-branch, and collects the branch post-conditions using disjunction. The rules for the basic commands (here, the assignments $\mathsf{r} := 0$ and $\mathsf{r} := \mathsf{r} + 1$ and the lookup $\mathsf{x} := [\mathsf{x} + 1]$) are also the same as in SL, as these are already exact. The unfolding of the list is also done in the same way, as unfolding always preserves equivalence; note how the condition of the `if` statement determines the appropriate disjunct for the list predicate. The recursive function call is allowed to go through as it is used with measure $n-1$ (with the parts of the assertion representing the pre- and the post-condition highlighted).

The major difference between ISL/ESL and SL proofs is that we cannot lose information about the function parameters and local variables in the middle of the former. Therefore, we cannot simplify the assertions Q'_1 and Q'_2 further and cannot fold back the list predicate within the internal specification, as we would do in SL (cf. corresponding proof in [24]).

The most complex part of the proof sketch is the transition from the internal to the external post-condition, in which we have to somehow forget the local variables of the function, given that they must not spill out into the calling context. This is done by replacing them with fresh, existentially quantified logical variables, which in this case also allows us to use equivalence to fold back the list predicate and reach the target post-condition. The details of this transition, in which we denote $\mathsf{ret} = n \star \alpha = n$ by R, are as follows:

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

Observe that, since we are proving an EX specification, we are not allowed to cut paths. This means that the ISL proof of the analogous ISL specification of $\mathtt{LLen}(\mathsf{x})$ would be identical, noting that the use of equivalence would technically be replaced by backward consequence.

```
// Function is recursive and requires a measure: α = n
Γ(α) ⊢ ( x = x ⋆ list(x, n) ⋆ α = n )
        LLen(x) {
          // Transition from external to internal pre-condition: initialise locals to null
          ( x = x ⋆ list(x, n) ⋆ α = n ⋆ r = null )
          if (x = null) {
            ( x = x ⋆ list(x, n) ⋆ α = n ⋆ r = null ⋆ x = null )
            r := 0
            ( Q'_1 : x = x ⋆ list(x, n) ⋆ α = n ⋆ r = 0 ⋆ x = null )
          } else {
            ( x = x ⋆ list(x, n) ⋆ α = n ⋆ r = null ⋆ x ≠ null )
            // Unfold list(x, n) using the equivalence
            // ⊨ list(x, n) ⋆ x ≠ null ⇔ ∃v, x'. x ↦ v, x' ⋆ list(x', n − 1)
            ( ∃v, x' . x = x ⋆ x ↦ v, x' ⋆ list(x', n − 1) ⋆ α = n ⋆ r = null )
            x := [x + 1];
            ( ∃v, x'. x = x' ⋆ x ↦ v, x' ⋆ list(x', n − 1) ⋆ α = n ⋆ r = null )
            // As α − 1 < α, we can use the specification of LLen(x) with measure α − 1
            ( ∃v, x'. x = x' ⋆ x ↦ v, x' ⋆ list(x', n − 1) ⋆ α − 1 = n − 1 ⋆ r = null )
            r := LLen(x);
            ( ∃v, x'. x = x' ⋆ x ↦ v, x' ⋆ list(x', n − 1) ⋆ α − 1 = n − 1 ⋆ r = n − 1 )
            r := r + 1
            ( Q'_2 : ∃v, x'. x = x' ⋆ x ↦ v, x' ⋆ list(x', n − 1) ⋆ α − 1 = n − 1 ⋆ r = n )
          };
          ( Q' : Q'_1 ∨ Q'_2 )
          return r
          ( Q' ⋆ ret = r )
          // Transition from internal to external post-condition given in text
        }
        ( list(x, n) ⋆ ret = n ⋆ α = n )
```

Figure 1 ESL proof sketch: $(\mathsf{x} = x \star \mathsf{list}(x, n))$ `LLen(x)` $(ok : \mathsf{list}(x, n) \star \mathsf{ret} = n)$.

List-insert in ESL: Intuition. The list-length function, `LLen(x)`, is an example of an algorithm where the EX specification is analogous to the traditional OX specification. At times, however, ESL specifications have to be more complex. Consider, for example, the list-insert algorithm `LInsertFirst(x, v)`, which inserts the element `v` at the beginning of the list `x`. Its traditional OX specification is:

$$\{\mathsf{x} = x \star \mathsf{v} = v \star \mathsf{list}(x, vs)\}\ \texttt{LInsertFirst}(\mathsf{x}, \mathsf{v})\ \{\mathsf{list}(\mathsf{ret}, v : vs)\}$$

where $\mathsf{list}(x, vs)$ is the standard list predicate that exposes the values of the list:

$$\mathsf{list}(x, vs) \triangleq (x = \texttt{null} \star vs = [\,]) \vee (\exists v, x', vs'.\, x \mapsto v, x' \star \mathsf{list}(x', vs') \star vs = v : vs')$$

Using the counter-example approach to check if this specification is EX-valid, we easily see that it loses information: in particular, no end-state where x is not the *second* pointer in the returned list `ret` is reachable from the given pre-condition. Consequently, for EX validity, we are required to use the following, less abstract, ESL specification for `LInsertFirst`:

$$(\mathsf{x} = x \star \mathsf{v} = v \star \mathsf{list}(x, xs, vs))\ \texttt{LInsertFirst}(\mathsf{x}, \mathsf{v})\ (\mathsf{list}(\mathsf{ret}, \mathsf{ret} : xs, v : vs) \star \mathsf{listHead}(x, xs))$$

where $\mathsf{list}(x, xs, vs)$ is a predicate that exposes the internal pointers of a given list in addition to the values, and $\mathsf{listHead}(x, xs)$ states that the list xs starts with x.

Further Examples. In §5 and [24], we give many additional examples of ESL specifications and proofs to illustrate reasoning about list algorithms and binary trees, as well as language errors, mutual recursion, non-termination, and client programs.

3 The Programming Language

We introduce ESL using a simple programming language, the syntax of which is given below.

Language Syntax

$$
\begin{array}{rcl}
v \in \mathsf{Val} & ::= & n \in \mathsf{Nat} \mid b \in \mathsf{Bool} \mid s \in \mathsf{Str} \mid \mathtt{null} \mid \overline{v} \qquad \mathsf{x} \in \mathsf{PVar} \\
\mathsf{E} \in \mathsf{PExp} & ::= & v \mid \mathsf{x} \mid \mathsf{E} + \mathsf{E} \mid \mathsf{E} - \mathsf{E} \mid \ldots \mid \mathsf{E} = \mathsf{E} \mid \mathsf{E} < \mathsf{E} \mid \neg\, \mathsf{E} \mid \mathsf{E} \wedge \mathsf{E} \mid \ldots \mid \mathsf{E} : \mathsf{E} \mid \mathsf{E} \mathbin{@} \mathsf{E} \mid \ldots \\
\mathsf{C} \in \mathsf{Cmd} & ::= & \mathtt{skip} \mid \mathsf{x} := \mathsf{E} \mid \mathsf{x} := \mathtt{nondet} \mid \mathtt{error}(\mathsf{E}) \mid \mathtt{if}\ (\mathsf{E})\ \mathsf{C}\ \mathtt{else}\ \mathsf{C} \mid \mathtt{while}\ (\mathsf{E})\ \mathsf{C} \mid \mathsf{C}; \mathsf{C} \mid \\
 & & \mathsf{y} := f(\vec{\mathsf{E}}) \mid \mathsf{x} := [\mathsf{E}] \mid [\mathsf{E}] := \mathsf{E} \mid \mathsf{x} := \mathtt{new}(\mathsf{E}) \mid \mathtt{free}(\mathsf{E})
\end{array}
$$

Values, $v \in \mathsf{Val}$, include: natural numbers, $n \in \mathsf{Nat}$; Booleans, $b \in \mathsf{Bool} \triangleq \{\mathsf{true}, \mathsf{false}\}$; strings, $s \in \mathsf{Str}$; a dedicated value `null`; and lists of values, $\overline{v} \in \mathsf{List}$. Expressions, $\mathsf{E} \in \mathsf{Exp}$, comprise values, program variables, $\mathsf{x} \in \mathsf{PVar}$, and various unary and binary operators (e.g., addition, equality, negation, conjunction, list prepending, and list concatenation). Commands comprise: the variable assignment; non-deterministic number generation; error raising; the `if` statement; the `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 $\mathtt{pv}(\mathsf{E})$ and $\mathtt{pv}(\mathsf{C})$ respectively, and the sets of modified variables for commands, denoted by $\mathtt{mod}(\mathsf{C})$, are defined in the standard way.

▶ **Definition 1** (Functions). *A function, denoted by* $f(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \mathtt{return}\ \mathsf{E}\,\}$*, comprises: a function identifier,* $f \in \mathsf{Str}$*; the function parameters,* $\vec{\mathsf{x}}$*, given by a list of distinct program variables; a function body,* $\mathsf{C} \in \mathsf{Cmd}$*; and a return expression,* $\mathsf{E} \in \mathsf{PExp}$*, with* $\mathtt{pv}(\mathsf{E}) \subseteq \{\vec{\mathsf{x}}\} \cup \mathtt{pv}(\mathsf{C})$*.*

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 2** (Function Implementation Contexts). *A function implementation context,* $\gamma : \mathsf{Str} \rightharpoonup_{fin} \mathsf{PVar}\ \mathsf{List} \times \mathsf{Cmd} \times \mathsf{PExp}$*, is a finite partial function from function identifiers to their implementations. For* $\gamma(f) = (\vec{\mathsf{x}}, \mathsf{C}, \mathsf{E})$*, we also write* $f(\vec{\mathsf{x}})\{\mathsf{C}; \mathtt{return}\ \mathsf{E}\} \in \gamma$*.*

We next 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** (Stores, Heaps, States). *Variable stores,* $s : \mathsf{PVar} \rightharpoonup_{\mathtt{fin}} \mathsf{Val}$*, are partial finite functions from program variables to values. Heaps,* $h : \mathsf{Nat} \rightharpoonup_{\mathtt{fin}} (\mathsf{Val} \uplus \varnothing)$*, are partial finite functions from natural numbers to values extended with a dedicated symbol* $\varnothing \notin \mathsf{Val}$*. Program states,* $\sigma = (s, h)$*, consist of a store and a heap.*

Heaps are used to model the memory, and the dedicated symbol $\varnothing \notin \mathsf{Val}$ is required for UX frame preservation[1] to hold (cf. Definition 10). In particular, $h(n) = v$ means that an allocated heap cell with address n contains the value v; and $h(n) = \varnothing$ means that a heap

[1] UX frame preservation means that if a program runs with a non-missing outcome to a given final state, then it also runs with the same outcome to an extended final state, with the extension (the *frame*) unaffected by the execution. From ISL [28], it is known that losing deallocation information breaks UX frame preservation; the solution is to keep track of deallocated cells, which we achieve by using $\varnothing$.

cell with address n has been deallocated [11–14, 28]. This linear memory model is used in much of the SL literature, including ISL [28]. Onward, $\emptyset$ denotes the empty heap, $h_1 \uplus h_2$ denotes heap disjoint union, and $h_1 \sharp h_2$ denotes that h_1 and h_2 are disjoint.

▶ **Definition 4** (Expression Evaluation). *The evaluation of an expression* E *with respect to a store* s*, denoted* $[\![\mathsf{E}]\!]_s$*, results in either a value or a dedicated symbol denoting an evaluation error,* $\lightning \notin \mathsf{Val}$*. Some illustrative cases are:*

$$[\![v]\!]_s = v \quad [\![\mathsf{x}]\!]_s = \begin{cases} s(\mathsf{x}), & \mathsf{x} \in \mathrm{dom}(s) \\ \lightning, & \textit{otherwise} \end{cases} \quad [\![\mathsf{E}_1 + \mathsf{E}_2]\!]_s = \begin{cases} [\![\mathsf{E}_1]\!]_s + [\![\mathsf{E}_2]\!]_s, & [\![\mathsf{E}_1]\!]_s, [\![\mathsf{E}_2]\!]_s \in \mathsf{Nat} \\ \lightning, & \textit{otherwise} \end{cases}$$

The big-step operational semantics uses judgements of the form $\sigma, \mathsf{C} \Downarrow_\gamma o : \sigma'$, 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 can either equal: ok (elided where possible), denoting a successful execution; err, denoting an execution faulting with a language error, or $miss$, denoting an execution faulting with a missing resource error.

▶ **Definition 5** (Operational Semantics). *The representative cases of the big-step operational semantics are given in Figure 2. The complete semantics is given in [24].*

The successful transitions 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 function arguments and the function locals 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, not available to the programmer. For simplicity of error messages, we assume to have a function $\mathsf{str} : \mathsf{PExp} \to \mathsf{Str}$, which serialises program expressions into strings. The faulting semantic transitions are split into *language errors*, which can be captured by program-logic reasoning, and *missing resource errors*, which cannot, as such errors break the frame property. Language errors arise due to, for example, expressions being incorrectly typed (e.g. `null` $+ 1$) or an attempt to access deallocated cells (that is, the use-after-free error). On the other hand, missing resource errors arise from accessing cells that are not present in memory.

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*, $x, y, z, \in \mathsf{LVar}$, distinct from program variables, and define the set of *logical expressions* as follows:

$$E \in \mathsf{LExp} \triangleq v \mid x \mid \mathsf{x} \mid E + E \mid E - E \mid ... \mid E = E \mid \neg\, E \mid E \wedge E \mid ... \mid E \cdot E \mid E : E \mid ...$$

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

$$\frac{[\![\mathsf{E}]\!]_s = v \quad s' = s[\mathsf{x} \to v]}{(s,h), \mathsf{x} := \mathsf{E} \Downarrow_\gamma (s',h)} \qquad \frac{n \in \mathbb{N} \quad s' = s[\mathsf{x} \to n]}{(s,h), \mathsf{x} := \texttt{nondet} \Downarrow_\gamma (s',h)} \qquad \frac{[\![\mathsf{E}]\!]_s = \mathsf{false}}{(s,h), \texttt{while}\ (\mathsf{E})\ \mathsf{C} \Downarrow_\gamma (s,h)}$$

$$\frac{[\![\mathsf{E}]\!]_s = \mathsf{true} \quad (s,h), \mathsf{C} \Downarrow_\gamma \sigma'' \quad \sigma'', \texttt{while}\ (\mathsf{E})\ \mathsf{C} \Downarrow_\gamma o : \sigma'}{(s,h), \texttt{while}\ (\mathsf{E})\ \mathsf{C} \Downarrow_\gamma o : \sigma'}$$

$$\frac{f(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \texttt{return}\ \mathsf{E}'\,\} \in \gamma \quad [\![\vec{\mathsf{E}}]\!]_s = \vec{v} \quad \mathtt{pv}(\mathsf{C}) \setminus \{\vec{\mathsf{x}}\} = \{\vec{\mathsf{z}}\} \quad s_p = \emptyset[\vec{\mathsf{x}} \to \vec{v}][\vec{\mathsf{z}} \to \texttt{null}] \quad (s_p,h), \mathsf{C} \Downarrow_\gamma (s_q,h') \quad [\![\mathsf{E}']\!]_{s_q} = v'}{(s,h), \mathsf{y} := f(\vec{\mathsf{E}}) \Downarrow_\gamma (s[\mathsf{y} \to v'], h')}$$

$$\frac{[\![\mathsf{E}]\!]_s = n \quad h(n) = v}{(s,h), \mathsf{x} := [\mathsf{E}] \Downarrow_\gamma (s[\mathsf{x} \to v], h)} \qquad \frac{[\![\mathsf{E}_1]\!]_s = n \quad h(n) \in \mathsf{Val} \quad [\![\mathsf{E}_2]\!]_s = v \quad h' = h[n \mapsto v]}{(s,h), [\mathsf{E}_1] := \mathsf{E}_2 \Downarrow_\gamma (s,h')}$$

$$\frac{(n' + i \notin \mathrm{dom}(h))|_{i=1}^{[\![\mathsf{E}]\!]_s - 1} \quad h' = h \uplus \{(n'+i \mapsto \texttt{null})|_{i=1}^{[\![\mathsf{E}]\!]_s - 1}\}}{(s,h), \mathsf{x} := \texttt{new}(\mathsf{E}) \Downarrow_\gamma (s[\mathsf{x} \to n'], h')} \qquad \frac{[\![\mathsf{E}]\!]_s = n \quad h(n) \in \mathsf{Val}}{(s,h), \texttt{free}(\mathsf{E}) \Downarrow_\gamma (s, h[n \mapsto \varnothing])}$$

$$\frac{[\![\mathsf{E}]\!]_s = \lightning \quad v_{err} = [\text{“ExprEval”}, \mathsf{str}(\mathsf{E})]}{(s,h), \mathsf{x} := [\mathsf{E}] \Downarrow_\gamma err : (s_{err}, h)} \qquad \frac{[\![\mathsf{E}]\!]_s = n \notin \mathrm{dom}(h) \quad v_{err} = [\text{“MissingCell”}, \mathsf{str}(\mathsf{E}), n]}{(s,h), \mathsf{x} := [\mathsf{E}] \Downarrow_\gamma miss : (s_{err}, h)}$$

$$\frac{h([\![\mathsf{E}]\!]_s) = \varnothing \quad v_{err} = [\text{“UseAfterFree”}, \mathsf{str}(\mathsf{E}_1), [\![\mathsf{E}]\!]_s]}{(s,h), \mathsf{x} := [\mathsf{E}] \Downarrow_\gamma err : (s_{err}, h)} \qquad \frac{[\![\mathsf{E}]\!]_s = v \quad v_{err} = [\text{“Error”}, v]}{(s,h), \texttt{error}(\mathsf{E}) \Downarrow_\gamma err : (s_{err}, h)}$$

Figure 2 Operational semantics (excerpt), with $s_{err} \triangleq s[\mathsf{err} \to v_{err}]$ and $\mathsf{str} : \mathsf{PExp} \to \mathsf{Str}$.

▶ **Definition 6** (Assertion Language). *The assertion language is defined as follows:*

$$\pi \in \mathsf{BAsrt} \triangleq E_1 = E_2 \mid E_1 < E_2 \mid E \in X \mid \ldots \mid \neg\pi \mid \pi_1 \Rightarrow \pi_2$$
$$P \in \mathsf{Asrt} \triangleq \pi \mid \mathsf{False} \mid P_1 \Rightarrow P_2 \mid \exists x.\, P \mid \mathsf{emp} \mid E_1 \mapsto E_2 \mid E \mapsto \varnothing \mid P_1 \star P_2 \mid \circledast_{E_1 \le x < E_2} P$$

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

Boolean assertions, $\pi \in \mathsf{BAsrt}$, lift Boolean logical expressions to assertions. Assertions, $P \in \mathsf{Asrt}$, contain Boolean assertions, standard first-order connectives and quantifiers, and spatial assertions. Spatial assertions include: the empty memory assertion, emp; the positive cell assertion, $E_1 \mapsto E_2$; the negative cell assertion, $E \mapsto \varnothing$ (as in [11–14] and denoted in ISL by $E \not\mapsto$ [28]), the separating conjunction (star); and its iteration (iterated star).

To define assertion satisfiability, we introduce *substitutions*, $\theta : \mathsf{LVar} \rightharpoonup_{\mathtt{fin}} \mathsf{Val}$, which are partial finite mappings from logical variables to values, extending expression evaluation of Definition 4 to $[\![E]\!]_{\theta,s}$ straightforwardly, with a new base case for logical variables:

$$[\![x]\!]_{\theta,s} = \theta(x), \text{if } x \in \mathsf{dom}(\theta) \qquad [\![x]\!]_{\theta,s} = \lightning, \text{if } x \notin \mathsf{dom}(\theta)$$

▶ **Definition 7** (Satisfiability). *The assertion satisfiability relation, denoted by* $\theta, \sigma \models P$, *is defined as follows:*

$$\begin{array}{lll}
\theta,(s,h) \models \pi & \Leftrightarrow & [\![\pi]\!]_{\theta,s} = \mathsf{true} \wedge h = \emptyset \\
\theta,(s,h) \models \mathsf{False} & \Leftrightarrow & \textit{never} \\
\theta,(s,h) \models P_1 \Rightarrow P_2 & \Leftrightarrow & \theta,(s,h) \models P_1 \Rightarrow \theta,(s,h) \models P_2 \\
\theta,(s,h) \models \exists x.\, P & \Leftrightarrow & \exists v \in \mathsf{Val}.\, \theta[x \mapsto v],(s,h) \models P \\
\theta,(s,h) \models \mathsf{emp} & \Leftrightarrow & h = \emptyset \\
\theta,(s,h) \models E_1 \mapsto E_2 & \Leftrightarrow & h = \{[\![E_1]\!]_{\theta,s} \mapsto [\![E_2]\!]_{\theta,s}\} \\
\theta,(s,h) \models E_1 \mapsto \varnothing & \Leftrightarrow & h = \{[\![E_1]\!]_{\theta,s} \mapsto \varnothing\} \\
\theta,(s,h) \models P_1 \star P_2 & \Leftrightarrow & \exists h_1, h_2.\, h = h_1 \uplus h_2 \wedge \theta,(s,h_1) \models P_1 \wedge \theta,(s,h_2) \models P_2 \\
\theta,(s,h) \models \circledast_{E_1 \le x < E_2} P & \Leftrightarrow & \exists h_i, \ldots, h_{k-1}.\, h = \uplus_{j=i}^{k-1} h_j \wedge \forall j.\, i \le j < k \Rightarrow \theta,(s,h_j) \models P[j/x] \\
 & & \textit{where } i = [\![E_1]\!]_{\theta,s},\ k = [\![E_2]\!]_{\theta,s},\ \textit{and } x \textit{ is not free in } E_1 \textit{ or } E_2.
\end{array}$$

Assertion satisfiability is defined in the standard way. For convenience, we choose Boolean assertions to be satisfiable only in the empty heap.

▶ **Definition 8** (Validity). *An assertion P is* valid*, denoted by $\models P$, iff $\forall \theta, \sigma.\ \theta, \sigma \models P$.*

4.2 Specifications

We define specifications for commands and functions, focussing in particular on external and internal function specifications and the relationship between them.

▶ **Definition 9.** Specifications, $t = (P)\ (ok : Q_{ok})\ (err : Q_{err}) \in \mathcal{S}pec$*, comprise a* pre-condition*, P, a* success post-condition*, Q_{ok}, and a* faulting post-condition*, Q_{err}.*

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

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

noting the use of $\mathcal{Q}$ for cases in which the post-condition details are not relevant. We use quadruples rather than triples since, even though the post-condition could be expressed as a disjunction of *ok*- and *err*-labelled assertions, we find the quadruple distinction helpful as compound commands (e.g. sequence) treat the two differently (cf. Figure 3).

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

▶ **Definition 10** (γ-Valid Specifications). *Given implementation context γ, command C, and specification $t = (P)\ (ok : Q_{ok})\ (err : Q_{err})$, t is* γ-valid *for C, denoted by $\gamma \models \mathsf{C} : t$ or $\gamma \models (P)\ \mathsf{C}\ (ok : Q_{ok})\ (err : Q_{err})$, if and only if:*

$$\begin{array}{l} \textit{// Frame-preserving over-approximating validity} \\ (\forall \theta, s, h, h_f, o, s', h''.\ \theta, (s, h) \models P \implies \\ \quad (s, h \uplus h_f), \mathsf{C} \Downarrow_\gamma o : (s', h'') \implies (o \neq \mathit{miss} \wedge \exists h'.\ h'' = h' \uplus h_f \wedge \theta, (s', h') \models Q_o)) \wedge \\ \textit{// Frame-preserving under-approximating validity} \\ (\forall \theta, s', h', h_f, o.\ \theta, (s', h') \models Q_o \implies h_f \sharp h' \implies \\ \quad (\exists s, h.\ \theta, (s, h) \models P \ \wedge\ (s, h \uplus h_f), \mathsf{C} \Downarrow_\gamma o : (s', h' \uplus h_f))) \end{array}$$

Observe that the outcome o can either be success or a language error; it cannot be a missing resource error as this would break UX 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 with both post-conditions False, then the execution of the command is guaranteed to not terminate if executed from a state satisfying the pre-condition. Were the semantics incomplete (for example, if it did not reason about errors), then such a specification might also indicate the absence of a semantic transition.

Compared to traditional OX reasoning, UX reasoning brings additional complexity to proofs of function specifications. To handle this complexity, we introduce two types of function 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 [25]. We use these in subsequent sections to show that ESL exhibits function compositionality.

▶ **Definition 11** (External Specifications). *A specification* $(P)\ (ok : Q_{ok})\ (err : Q_{err})$ *is an* external function specification *if and only if:*

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

The set of external specifications is denoted by $\mathcal{ES}pec$*.*

▶ **Definition 12** (Function Specification Contexts). *A function specification context,* $\Gamma : \mathsf{Fid} \rightharpoonup_{fin} \mathcal{P}(\mathcal{ES}pec)$*, is a finite partial function from function identifiers to a set of external specifications, with the more familiar notation* $(\vec{\mathsf{x}} = \vec{x} \star P)\ f(\vec{\mathsf{x}})\ (ok : Q_{ok})\ (err : Q_{err}) \in \Gamma$ *at times used in place of* $(\vec{\mathsf{x}} = \vec{x} \star P)\ (ok : Q_{ok})\ (err : Q_{err}) \in \Gamma(f)$*.*

The constraints on external specifications are well-known from OX logics and follow the usual scoping of function parameters and local variables, which are limited to the function body: the pre-conditions only contain the function parameters, $\vec{\mathsf{x}}$; and the post-conditions may only have the (dedicated) program variables ret or err, which hold, respectively, the return value on successful termination or the error value on faulting termination.

Internal function specifications are more interesting for exact and UX than for OX reasoning. The internal pre-condition is straightforward, extending the external pre-condition by instantiating the local variables to `null`. The internal post-condition must therefore include information about the parameters and local variables, as the internal specification cannot lose information. This means that the connection between internal and external post-conditions is subtle, given the constraints on the latter. 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 has to be equivalent to an internal one in which the parameters and local variables of the internal post-condition have been replaced by fresh existentially quantified logical variables.

▶ **Definition 13** (Internalisation). *Given implementation context* γ *and function* $f \in \mathrm{dom}(\gamma)$*, a function specification internalisation,* $Int_{\gamma,f} : \mathcal{ES}pec \to \mathcal{P}(\mathcal{S}pec)$*, is defined as follows:*

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

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

This approach also works for SL and ISL as well (with $\Leftarrow$ instead of $\Leftrightarrow$ for the post-conditions for SL, and $\Rightarrow$ instead of $\Leftrightarrow$ for ISL). It is not strictly necessary for SL, however, as information about program variables can be forgotten in the internal post-conditions before the transition to the external post-condition.

▶ **Definition 14** (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 15** (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*

$$\begin{array}{l}\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\gamma) \wedge \\ \quad (\forall f, \vec{\mathsf{x}}, \mathsf{C}, \mathsf{E}.\ f(\vec{\mathsf{x}})\{\mathsf{C}; \texttt{return}\ \mathsf{E}\} \in \gamma \implies (\forall t.\ t \in \Gamma(f) \implies \exists t' \in \mathit{Int}_{\gamma,f}(t).\ \gamma \models \mathsf{C} : t'))\end{array}$$

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 16** (Γ-Valid Specifications). *Given a specification context Γ, a command C, and a specification $t = (P)\ (\mathcal{Q})$, the specification t is* Γ-valid *for command C, written $\Gamma \models \mathsf{C} : t$ or $\Gamma \models (P)\ \mathsf{C}\ (\mathcal{Q})$, if and only if $\forall \gamma.\ \models (\gamma, \Gamma) \implies \gamma \models (P)\ \mathsf{C}\ (\mathcal{Q})$.*

4.3 Program Logic

We give the representative ESL proof rules in Figure 3 and all in [24]. We introduce and discuss in detail the function-related rules, given for the first time in a UX setting. We denote the repetition of the pre-condition in the post-condition by pre. When reading the rules, it is important to remember that we *must not drop paths and must not lose information*. The judgement $\Gamma \vdash (P)\ \mathsf{C}\ (ok : Q_{ok})\ (err : Q_{err})$ means that the specification t is *derivable* for a command C given the specifications recorded in Γ.

The basic command rules are fairly straightforward. The [NONDET] rule existentially quantifies the generated value via $\mathsf{x} \in \mathbb{N}$ to capture all paths, in contrast with the RHL [8] and ISL [28] rules, which explicitly choose one value to describe one path. The $E' \in \mathsf{Val}$ in the post-condition is necessary as we know that E' evaluates to a value from the pre-condition and cannot lose information; the same principle applies to many other rules. The [ASSIGN] rule requires that the evaluation of E does not fault in the pre-condition via $\mathsf{E} \in \mathsf{Val}$. Strictly speaking, we should have an additional case in which the assigned variable is not in the store. To avoid this clutter, we instead assume that program variables are always in the store as we are analysing function bodies and, in our programming language, 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 $\mathsf{E} \notin \mathsf{Val}$), expressions are of the incorrect type, or memory is accessed after it has been freed (e.g. [LOOKUP-ERR-USE-AFTER-FREE] rule, using $\mathsf{E} \mapsto \varnothing$). 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. One interesting aspect of this rule is what happens when C_1 only throws an error or does not terminate, meaning that $R = \mathsf{False}$. In both those cases, given the exactness of the rules, it has to be that $Q_{ok} = Q^2_{err} = \mathsf{False}$, and the post-condition of the sequence becomes $(ok : \mathsf{False})\ (err : Q^1_{err} \vee \mathsf{False})$, meaning that, if C_1 only throws an error (that is, $Q^1_{err} \neq \mathsf{False}$) then that is the only error that can come out of the sequence, and if C_1 does not terminate (that is, $Q^1_{err} = \mathsf{False}$) then the sequence does not terminate either.

The while rule is an adaptation of the RHL while rule [8], generalising the invariant of the SL while rule with two natural-number-indexed families of variants, P_i and Q_i, which explicitly maintain the iteration index. Note how the i in the premise is a meta-variable representing a natural number, which in the conclusion gets substituted for an existentially quantified logical variable; a similar principle will be applied later when dealing with environment extension. Interestingly, this rule does not require adjustment to reason about non-termination.

$$\text{SKIP}\quad \Gamma \vdash (\mathsf{emp})\ \texttt{skip}\ (\mathsf{emp})$$

$$\text{NONDET}\quad \frac{\mathsf{x} \notin \mathtt{pv}(E') \qquad Q \triangleq E' \in \mathsf{Val} \star \mathsf{x} \in \mathbb{N}}{\Gamma \vdash (\mathsf{x} = E')\ \mathsf{x} := \texttt{nondet}\ (Q)}$$

$$\text{ASSIGN}\quad \frac{\mathsf{x} \notin \mathtt{pv}(E') \qquad Q \triangleq E' \in \mathsf{Val} \star \mathsf{x} = \mathsf{E}[E'/\mathsf{x}]}{\Gamma \vdash (\mathsf{x} = E' \star \mathsf{E} \in \mathsf{Val})\ \mathsf{x} := \mathsf{E}\ (Q)}$$

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

$$\text{MUTATE}\quad \frac{Q \triangleq \mathsf{E}_1 \mapsto \mathsf{E}_2 \star E \in \mathsf{Val}}{\Gamma \vdash (\mathsf{E}_1 \mapsto E \star \mathsf{E}_2 \in \mathsf{Val})\ [\mathsf{E}_1] := \mathsf{E}_2\ (Q)}$$

$$\text{NEW}\quad \frac{\mathsf{x} \notin \mathtt{pv}(E') \qquad Q \triangleq E' \in \mathsf{Val} \star \circledast_{0 \le i < \mathsf{E}[E'/\mathsf{x}]}((\mathsf{x}+i) \mapsto \texttt{null})}{\Gamma \vdash (\mathsf{x} = E' \star \mathsf{E} \in \mathbb{N})\ \mathsf{x} := \texttt{new}(\mathsf{E})\ (ok : Q)}$$

$$\text{ERROR}\quad \frac{E_{err} \triangleq [\text{“Error”}, \mathsf{E}]}{\Gamma \vdash (\mathsf{E} \in \mathsf{Val})\ \texttt{error}(\mathsf{E})\ (err : \mathsf{err} = E_{err})}$$

$$\text{FREE}\quad \frac{Q \triangleq E' \in \mathsf{Val} \star \mathsf{E} \mapsto \varnothing}{\Gamma \vdash (\mathsf{E} \mapsto E')\ \texttt{free}(\mathsf{E})\ (ok : Q)}$$

$$\text{LOOKUP-ERR-VAL}\quad \frac{P \triangleq \mathsf{x} = E' \star E \notin \mathsf{Val} \qquad E_{err} \triangleq [\text{“ExprEval”}, \mathsf{str}(\mathsf{E})]}{\Gamma \vdash (P)\ \mathsf{x} := [\mathsf{E}]\ (err : Q^*_{err})}$$

$$\text{LOOKUP-ERR-USE-AFTER-FREE}\quad \frac{P \triangleq \mathsf{x} = E' \star \mathsf{E} \mapsto \varnothing \qquad E_{err} \triangleq [\text{“UseAfterFree”}, \mathsf{str}(\mathsf{E}), \mathsf{E}]}{\Gamma \vdash (P)\ \mathsf{x} := [\mathsf{E}]\ (err : Q^*_{err})}$$

$$\text{IF-THEN}\quad \frac{\mathsf{C} \triangleq \texttt{if}\ (\mathsf{E})\ \mathsf{C}_1\ \texttt{else}\ \mathsf{C}_2 \qquad \Gamma \vdash (P \star \mathsf{E})\ \mathsf{C}_1\ (\mathcal{Q})}{\Gamma \vdash (P \star \mathsf{E})\ \mathsf{C}\ (\mathcal{Q})}$$

$$\text{IF-ELSE}\quad \frac{\mathsf{C} \triangleq \texttt{if}\ (\mathsf{E})\ \mathsf{C}_1\ \texttt{else}\ \mathsf{C}_2 \qquad \Gamma \vdash (P \star \neg\mathsf{E})\ \mathsf{C}_2\ (\mathcal{Q})}{\Gamma \vdash (P \star \neg\mathsf{E})\ \mathsf{C}\ (\mathcal{Q})}$$

$$\text{IF-ERR-VAL}\quad \frac{\mathsf{C} \triangleq \texttt{if}\ (\mathsf{E})\ \mathsf{C}_1\ \texttt{else}\ \mathsf{C}_2 \qquad E_{err} \triangleq [\text{“ExprEval”}, \mathsf{str}(\mathsf{E})]}{\Gamma \vdash (P \star \mathsf{E} \notin \mathsf{Val})\ \mathsf{C}\ (err : Q^*_{err})}$$

$$\text{SEQ}\quad \frac{\Gamma \vdash (P)\ \mathsf{C}_1\ (ok : R)\ (err : Q^1_{err}) \qquad \Gamma \vdash (R)\ \mathsf{C}_2\ (ok : Q_{ok})\ (err : Q^2_{err})}{\Gamma \vdash (P)\ \mathsf{C}_1;\ \mathsf{C}_2\ (ok : Q_{ok})\ (err : Q^1_{err} \vee Q^2_{err})}$$

$$\text{WHILE}\quad \frac{\forall i \in \mathbb{N}.\ \models P_i \Rightarrow \mathsf{E} \in \mathbb{B} \star \mathsf{True} \qquad \forall i \in \mathbb{N}.\ \Gamma \vdash (P_i \star \mathsf{E})\ \mathsf{C}\ (ok : P_{i+1})\ (err : Q_i)}{\Gamma \vdash (P_0)\ \texttt{while}\ (\mathsf{E})\ \mathsf{C}\ (ok : \neg\mathsf{E} \star \exists i.\, P_i)\ (err : \exists i.\, Q_i)}$$

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

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

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

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

Figure 3 ESL proof rules (excerpt), with $Q^*_{err} = (\mathsf{pre} \star \mathsf{err} = E_{err})$.

The structural rules are not surprising, with equivalence replacing the forward/backward consequence of OX/UX reasoning and with frame, existential introduction, 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 that there, however, is no sound conjunction rule, as the conjunction rules of SL and ISL cannot be combined in ESL, since conjunction does not distribute over the star in both directions, breaking frame preservation.

Function Call. We discuss the ESL function-call rule in detail, creating it starting from the standard OX-sound SL rule, adapted for quadruples:

$$\frac{\{\vec{\mathsf{x}} = \vec{x} \star P\}\ f(\vec{\mathsf{x}})\ \{ok : Q_{ok}\}\ \{err : Q_{err}\} \in \Gamma \qquad \mathsf{y} \notin \mathtt{pv}(E_y)}{\Gamma \vdash \left\{\mathsf{y} = E_y \star \vec{\mathsf{E}} = \vec{x} \star P\right\}\ \mathsf{y} := f(\vec{\mathsf{E}})\ \{ok : Q_{ok}[\mathsf{y}/\mathtt{ret}]\}\ \{err : \mathsf{y} = E_y \star Q_{err}\}}$$

In order to make this rule UX-sound, we only have to ensure that no information from the pre-condition is lost in the post-conditions. In particular, we have to remember:

- that the evaluation of E_y does not fault, captured by $E_y \in \mathsf{Val}$ and needed in the success post-condition only, as it is already implied by the error post-condition; and
- that $\vec{\mathsf{E}} = \vec{x}$ holds (with the substitution $[E_y/\mathsf{y}]$ needed in the success post-condition as the value of y may change if the function call succeeds);

bringing us to the ESL function call rule:

$$\frac{\begin{array}{c}(\vec{\mathsf{x}} = \vec{x} \star P)\ f(\vec{\mathsf{x}})\ (ok : Q_{ok})\ (err : Q_{err}) \in \Gamma \qquad \mathsf{y} \notin \mathtt{pv}(E_y) \\ Q'_{ok} \triangleq E_y \in \mathsf{Val} \star \vec{\mathsf{E}}[E_y/\mathsf{y}] = \vec{x} \star Q_{ok}[\mathsf{y}/\mathtt{ret}] \qquad Q'_{err} \triangleq \mathsf{y} = E_y \star \vec{\mathsf{E}} = \vec{x} \star Q_{err}\end{array}}{\Gamma \vdash \left(\mathsf{y} = E_y \star \vec{\mathsf{E}} = \vec{x} \star P\right)\ \mathsf{y} := f(\vec{\mathsf{E}})\ (ok : Q'_{ok})\ (err : Q'_{err})}$$

Environment Formation. Whereas the ESL function-call rule does not deviate substantially from its OX counterpart, the environment formation rules illustrate the difference in complexity between OX and UX function compositionality. These rules use the judgement $\vdash (\gamma, \Gamma)$ to state that the environment (γ, Γ) is *well-formed*. The base case, $\vdash (\emptyset, \emptyset)$, is trivial and the same as for SL, stating that the environment consisting of an empty implementation context and an empty specification context is well-formed. For illustrative purposes, we give a simplified version of the extension rule, extending the environment with a single, possibly recursive, function. The full rule, which extends the environment with a group of mutually recursive functions, is given in [24]. We start from the corresponding OX-sound rule from SL:

$$\text{ENV-EXTEND-SL}\quad \frac{\vdash (\gamma, \Gamma) \quad f \notin \mathrm{dom}(\gamma) \quad \gamma' = \gamma[f \mapsto (\vec{\mathsf{x}}, \mathsf{C}, \mathsf{E})] \quad \Gamma' = \Gamma[f \mapsto \{t\}] \quad \exists t' \in \mathit{Int}_{\gamma', f}(t).\ \Gamma' \vdash \mathsf{C} : t'}{\vdash (\gamma', \Gamma')}$$

which states that a well-formed environment (γ, Γ) can be extended with a given function f and its external specification t to (γ', Γ') if some corresponding internal specification of f can be proven for the body of f under the extended specification context Γ'.[2] Note that using Γ' means that a specification can be used to prove itself, which is sound in SL but unsound in ISL: specifically, it would allow us to prove UX-invalid specifications of non-terminating functions. For example, we would be able to prove that the function $\mathtt{f}()\ \{\ \mathsf{r} := \mathtt{f}();\ \mathtt{return}\ \mathsf{r}\ \}$ satisfies the EX-valid specification $(\mathsf{emp})\ \mathtt{f}()\ (\mathsf{False})$, but also the EX-invalid specification $(\mathsf{emp})\ \mathtt{f}()\ (\mathsf{ret} = 42)$. The latter is vacuously OX-valid as there are no terminating executions, but when considered from the UX viewpoint, it implies the existence of an execution path from the pre- to the post-condition, contradicting the non-termination of $\mathtt{f}$.

Therefore, to be soundly usable in UX reasoning, specifications with satisfiable post-conditions (onward: terminating specifications) must come with a mechanism that disallows the above counter-example. We achieve this by following a standard approach for reasoning about termination [7, 9], based on decreasing measures on well-ordered sets. In particular, we require the terminating specification to be proven, $t \triangleq (P)\ (ok : Q_{ok})\ (err : Q_{err})$ to be extended with a *measure* $\alpha \in \mathbb{N}$, denoting this extension by $t(\alpha)$:[3]

$$t(\alpha) \triangleq (P \star \alpha = E_\mu)\ (ok : Q_{ok} \star \alpha = E_\mu)\ (err : Q_{err} \star \alpha = E_\mu)$$

where E_μ is a logical expression describing how the measure is computed. Then, we prove that $t(\alpha)$ holds for every specific α, assuming that recursive calls to f can only use the terminating

[2] Internalisation is normally omitted in SL as forward consequence allows information about program variables to be lost, making internal and external post-conditions of SL specifications almost the same.

[3] We can extend the measure beyond natural numbers to computable ordinals, $\mathcal{O} \triangleq \omega_1^{\mathrm{CK}}$, allowing us to reason about a broader set of functions, such as those with non-deterministic nested recursion (cf. [24]).

specifications $t(\beta)$ of a measure β *strictly smaller* than α. This restriction is standard and, if the proof succeeds, ensures that f has at least one terminating execution. Also, it disallows the above-mentioned counter-example, as no measure given in the pre-condition would be able to decrease before the recursive call. Importantly, the measure is only a tool required for proving specification validity and once this proof has been completed, the specification without the measure is added to Γ and can be used in proofs of client code.

In addition, we incorporate reasoning about non-terminating specifications (NT-specifications). This is relevant in situations in which the operational semantics of the analysed language is complete, which allows non-termination to be captured using the post-condition False. As NT-specifications are vacuously UX-sound, our focus is on ensuring their OX-soundness, which we do by again imposing a measure α, but allowing recursive calls for a measure β *smaller or equal* than α, that is, for an NT-specification to be used to prove itself. This, for example, allows for a proof of the specification (emp) `f() { r := f(); return r }` (False) by choosing a constant measure α. The ESL environment extension rule, therefore, is as follows:

ENV-EXTEND

// Assume valid environment, extend implementation context with new function f

$\vdash (\gamma, \Gamma) \qquad f \notin \mathrm{dom}(\gamma) \qquad \gamma' = \gamma[f \mapsto (\vec{x}, \mathsf{C}, \mathsf{E})]$

// Extend the specifications of f with a measure α

$t \triangleq (P)\ (ok : Q_{ok})\ (err : Q_{err}) \quad t_\infty \triangleq (P_\infty)\ (\mathsf{False}) \quad t_\infty(\alpha) \triangleq (P_\infty \star \alpha = E_\mu)\ (\mathsf{False})$

$t(\alpha) \triangleq (P \star \alpha = E_\mu)\ (ok : Q_{ok} \star \alpha = E_\mu)\ (err : Q_{err} \star \alpha = E_\mu)$

// Construct $\Gamma(\alpha)$: assume t for measure $\beta < \alpha$ and t_∞ for measure $\beta \leq \alpha$

$\Gamma(\alpha) = \Gamma[f \mapsto \{t(\beta) \mid \beta < \alpha\} \cup \{t_\infty \mid \beta \leq \alpha\}]$

// For every α, prove internal specifications of f corresponding to t and t_∞

$\forall \alpha.\, \exists t' \in \mathit{Int}_{\gamma', f}(t(\alpha)).\ \Gamma(\alpha) \vdash \mathsf{C} : t' \qquad \forall \alpha.\, \exists t' \in \mathit{Int}_{\gamma', f}(t_\infty(\alpha)).\ \Gamma(\alpha) \vdash \mathsf{C} : t'$

// Extend Γ with t and t_∞

$\Gamma' := \Gamma[f \mapsto \{t, t_\infty\}]$

$\vdash (\gamma', \Gamma')$

4.4 Soundness

We state the soundness results for ESL and give intuition about the proofs; the full proofs can be found in [24].

▶ **Theorem 17.** *Any derivable specification is valid:* $\Gamma \vdash (P)\ \mathsf{C}\ (Q) \implies \Gamma \models (P)\ \mathsf{C}\ (Q)$.

Proof. By induction on $\Gamma \vdash (P)\ \mathsf{C}\ (Q)$. Most cases are straightforward; the [FUN-CALL] rule obtains a valid specification for the function body from the validity of the environment. ◀

▶ **Theorem 18.** *Any well-formed environment is valid:* $\vdash (\gamma, \Gamma) \implies \models (\gamma, \Gamma)$.

Proof. At the core of the proof is a lemma stating that $\models (\gamma, \Gamma) \implies (\forall \alpha. \models (\gamma', \Gamma(\alpha)))$, where γ' and $\Gamma(\alpha)$ have been obtained from γ and Γ as per [ENV-EXTEND]. Using this lemma, we derive the desired $\models (\gamma', \Gamma')$, where Γ' is obtained from Γ and $\Gamma(\alpha)$ as per [ENV-EXTEND]. The proof of this lemma is done by transfinite induction on α and has the standard zero, successor, and limit ordinal cases. We outline the proof for the case in which a single, possibly recursive, function f with body C_f is added; the generalisation to n mutually recursive functions is straightforward and can be seen in [24].

In all three cases, the soundness of all specifications except the NT-specification with the highest considered ordinal follows from the inductive hypothesis. This remaining NT-specification is vacuously UX-valid, meaning that we only need to prove its OX-validity. For this, we use a form of fixpoint induction called Scott induction (see, e.g., Winskel [31]), required when specifications can be used to prove themselves (e.g. any SL specification).

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 $h(\mathsf{C}) \triangleq \mathsf{C}_f[\mathsf{C}, \gamma', f]$, 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 $\mathbb{C}_{\gamma'}$: $g([\mathsf{C}]) := [h(\mathsf{C})]$. 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 $\mathsf{C}_f \in \mathit{lfp}(g)$, to obtain that $[\mathsf{C}_f] \in S^\alpha$. From there, we are finally able to prove that $\models (\gamma', \Gamma(\alpha))$. ◀

These two theorems, to the best of our knowledge, are the first to demonstrate sound function compositionality for UX logics. Previous work on UX logics [25, 28, 29] used function specifications in examples but did not include rules in the logic for calling functions and managing a function specification environment. Program logics that do not include function rules and function specification environments in effect delegate soundness responsibilities to the meta-logic within which they are embedded. This might be appropriate in some contexts, such as in interactive theorem provers, whose meta-logic is reliable. Charguéraud's clean-slate tutorial SL implementation in Coq [6], for example, does not provide either function call rules or program-logic-level infrastructure for a function specification environment; instead, it relies on Coq's induction mechanism and definitional mechanism to use and store function specifications. However, when implementing an SL/ISL/ESL-based tool in a mainstream non-ITP language, such as C++ or OCaml, no reliable meta-logic that can act as a safety net is available. This is particularly concerning for UX logics, which require complex rules for handling functions, including forgetting information about function-local mutable variables. In ESL, the handling of functions is fully internalised into the logic, no meta-logic facilities are required to handle function calls, and the program-logic-level facilities of ESL for handling functions are validated by the soundness proof.

The proof of Theorem 18 can be adjusted for ISL: the function call rule would remain the same, and [ENV-EXTEND] would not include NT-specifications, removing the need for Scott induction. On the other hand, the Scott induction itself could be easily adapted for SL.

5 Examples: ESL in Practice

We demonstrate how to use ESL to specify and verify correctness and incorrectness properties of data-structure algorithms, focussing on singly-linked lists and binary search trees.

We investigate, for the first time, the use of abstract predicates in a UX program logic, decoupling abstraction from over-approximation. Our findings show that UX/EX specifications can soundly incorporate abstraction, but also that it, ultimately, cannot be used as freely as in the OX setting. Firstly, since UX reasoning cannot lose information, not all algorithms can be UX-specified at all levels of abstraction, and hence sometimes specifications have to be less abstract than in OX reasoning. Secondly, because specifications are only composable when expressed at the same level of abstraction, specifications of a library client have to be written at the "least common level of abstraction" of the specifications of all of the library functions that the client calls.

Building on §2, we give further intuition on how to think informally about UX/EX specifications using a number of list algorithms and predicates describing lists with various degrees of abstraction (§5.1, §5.2), more detail on how to write formal ESL proofs (§5.3), and examples of ESL reasoning for binary-search-tree algorithms (§5.4).

5.1 List Predicates

We implement singly-linked lists in the standard way: every list consists of two contiguous cells in the heap (denoted $x \mapsto a, b$, meaning $x \mapsto a \star x + 1 \mapsto b$), 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 a number of list predicates:

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

These predicates expose different parts of the list structure in their parameters, *hiding* the rest via existential quantification: the $\mathsf{list}(x)$ predicate hides all information about the represented mathematical list, just declaring that there is a singly-linked list at address x; the $\mathsf{list}(x, n)$ predicate hides the internal node addresses and values, exposing the list length via the parameter n; the $\mathsf{list}(x, xs)$ predicate hides information about the values of the mathematical list, exposing the internal addresses of the list via the parameter xs; the $\mathsf{list}(x, vs)$ predicate hides information about the internal addresses, exposing the list's values via the parameter vs; and the list predicate $\mathsf{list}(x, xs, vs)$ hides nothing, exposing the entire node-value structure via the parameters xs and vs. These predicates are related to each other via logical equivalence; for example, it holds that:

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

5.2 Writing UX/EX Abstract Specifications

We consider a number of list algorithms, described in words, and guide the reader on how to write correct UX/EX specifications for these algorithms using the list abstractions given in the previous section (§5.1), comparing how the UX/EX approach and specifications differ from their OX counterparts. We provide detailed proofs and implementations for each type of algorithm (iterative/recursive, allocating/deallocating, pure/mutative, etc.) in [24].

An important point is to understand how to look for counter-examples to a given specification: from the definition of OX validity, it follows that breaking OX reasoning amounts to "finding a state in the pre-condition (pre-model) for which the execution of f terminates and does not end in a state in the post-condition"; from the definition of UX validity, breaking UX reasoning means "finding a model of the post-condition (post-model) not reachable by execution of f from any state in the pre-condition"; and from the definition of EX validity, it follows that breaking EX reasoning means breaking either OX or UX reasoning. In addition, it is useful to remember that, for breaking UX validity, it is sufficient to find information known in the pre-condition but lost in the post-condition.

Length. We first revisit the list-length function $\texttt{LLen}(\mathsf{x})$, which takes a list at address x, does not modify it, and returns its length. In §2, we have shown that it satisfies the exact specification $(\mathsf{x} = x \star \mathsf{list}(x, n))\ \ \texttt{LLen}(\mathsf{x})\ \ (\mathsf{list}(x, n) \star \mathsf{ret} = n)$ and observed that

(O1) abstraction does not always equal over-approximation.

Using similar reasoning, we can come to the conclusion that the following, less abstract specifications for $\texttt{LLen}(\mathsf{x})$ are also EX-valid:

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

On the other hand, if we consider the following OX-valid specification:

$$\{\mathsf{x} = x \star \mathsf{list}(x)\}\ \ \texttt{LLen}(\mathsf{x})\ \ \{\exists n \in \mathbb{N}.\ \mathsf{list}(x) \star \mathsf{ret} = n\}$$

we see that it is not UX-valid as the post-condition does not connect the return value to the list. In particular, if we choose a post-model for $\mathsf{list}(x)$ that has length 2, but then choose, for example, $\mathsf{ret} = 42$, we run into a problem: as the algorithm does not modify the list, we have to choose the same model of $\mathsf{list}(x)$ for the pre-model to have a chance of reaching the post-model, but then the algorithm will return 2, not 42, meaning that this specification is indeed not UX/EX-valid. From this discussion, we observe that:

(O2) in valid UX/EX specifications, data-structure abstractions used in a post-condition must expose enough information to capture the behaviour of the function being specified with respect to the information given in the pre-condition.

Note that, given a specification less abstract than strictly needed, one can obtain more abstract ones by using validity-preserving transformations on specifications that correspond to the structural rules of the logic. We refer to these as *admissible* transformations, give the ones for existential introduction and equivalence below, and the rest in [24]:

$$\text{ADM-EXISTS}\quad \frac{\Gamma \models (\vec{\mathsf{x}} = \vec{x} \star P)\ f(\vec{\mathsf{x}})\ (ok : Q_{ok})\ (err : Q_{err}) \qquad y \notin \vec{x}}{\Gamma \models (\vec{\mathsf{x}} = \vec{x} \star \exists y.\ P)\ f(\vec{\mathsf{x}})\ (ok : \exists y.\ Q_{ok})\ (err : err : \exists y.\ Q_{err})}$$

$$\text{ADM-EQUIV}\quad \frac{\Gamma \models (\vec{\mathsf{x}} = \vec{x} \star P')\ f(\vec{\mathsf{x}})\ (ok : Q'_{ok})\ (err : Q'_{err}) \qquad \models P', Q'_{ok}, Q'_{err} \Leftrightarrow P, Q_{ok}, Q_{err}}{\Gamma \models (\vec{\mathsf{x}} = \vec{x} \star P)\ f(\vec{\mathsf{x}})\ (ok : Q_{ok})\ (err : Q_{err})}$$

For list-length, starting from the least abstract specification using $\mathsf{list}(x, xs, vs)$, we can derive, for example, the specification using $\mathsf{list}(x, n)$, as follows:

$$\begin{array}{ll} & (\mathsf{x} = x \star \mathsf{list}(x, xs, vs))\ \ \texttt{LLen}(\mathsf{x})\ \ (\mathsf{list}(x, xs, vs) \star \mathsf{ret} = |xs|) \\ [\text{ADM-EXISTS}] & (\exists vs.\ \mathsf{x} = x \star \mathsf{list}(x, xs, vs))\ \ \texttt{LLen}(\mathsf{x})\ \ (\exists vs.\ \mathsf{list}(x, xs, vs) \star \mathsf{ret} = |xs|) \\ [\text{ADM-EXISTS}] & (\exists xs, vs.\ \mathsf{x} = x \star \mathsf{list}(x, xs, vs))\ \ \texttt{LLen}(\mathsf{x})\ \ (\exists xs, vs.\ \mathsf{list}(x, xs, vs) \star \mathsf{ret} = |xs|) \\ [\text{ADM-EQUIV}] & (\mathsf{x} = x \star \mathsf{list}(x, n))\ \ \texttt{LLen}(\mathsf{x})\ \ (\mathsf{list}(x, n) \star \mathsf{ret} = n) \end{array}$$

Interestingly, from one more application of [ADM-EXISTS] and [ADM-EQUIV], we can derive

$$(\mathsf{x} = x \star \mathsf{list}(x))\ \ \texttt{LLen}(\mathsf{x})\ \ (\exists n.\ \mathsf{list}(x, n) \star \mathsf{ret} = n)$$

which further illustrates observation (O2), in that even though the pre-condition does not talk about the length of the list, the post-condition has to expose it because the function output depends on it, and hence the post-condition must connect up the return value to the length of the list, here by an existentially quantified variable.

This approach of deriving abstract specifications can be used in general for working with ESL: for a given algorithm, first prove the least abstract specification, which exposes all details, and then adjust the degree of abstraction to fit the needs of the client code. We discuss this further in the upcoming paragraph on reasoning about client programs.

Membership. Next, we consider the list-membership function `LMem`(x, v), which takes a list at address x, does not modify it, and returns true if v is in the list, and false otherwise. Given (O2) and the fact that the function output depends on the values in the list, we understand that, for its UX/EX specification, we should be using a list abstraction that exposes at least the values, that is, $\mathsf{list}(x, vs)$ or $\mathsf{list}(x, xs, vs)$. The corresponding specifications are:

$$(\mathsf{x} = x \star \mathsf{v} = v \star \mathsf{list}(x, vs))\ \ \texttt{LMem}(\mathsf{x}, \mathsf{v})\ \ (\mathsf{list}(x, vs) \star \mathsf{ret} = (v \in vs))$$
$$(\mathsf{x} = x \star \mathsf{v} = v \star \mathsf{list}(x, xs, vs))\ \ \texttt{LMem}(\mathsf{x}, \mathsf{v})\ \ (\mathsf{list}(x, xs, vs) \star \mathsf{ret} = (v \in vs))$$

and are proven similarly to list-length. We can check that a more abstract specification, say:

$$\{\mathsf{x} = x \star \mathsf{v} = v \star \mathsf{list}(x)\}\ \ \texttt{LMem}(\mathsf{x}, \mathsf{v})\ \ \{\exists b \in \mathbb{B}.\ \mathsf{list}(x) \star \mathsf{ret} = b\}$$

is not UX-valid, by choosing, as the post-model, b to be false and the list at x to contain v. As for list-length, since list-membership does not modify the list, we have to choose the same model of $\mathsf{list}(x)$ for the pre-model to have a chance of reaching the post-model, but then the algorithm will return true, not false, so this post-model is not reachable.

Swap-First-Two. Next, we consider the list-swap-first-two function `LSwapFirstTwo`(x), which takes a list at address x, swaps its first two values if the list is of sufficient length, returning `null`, and throws an error otherwise without modifying the list. Given (O2), to specify this function we need an abstraction that captures list length and, apparently, also the list values; for example, $\mathsf{list}(x, vs)$. As this function can throw errors, its full EX specification has to use the ESL quadruple, in which the two post-conditions are constrained with the corresponding, shaded, branching conditions:

$$\begin{array}{c}(\mathsf{x} = x \star \mathsf{list}(x, vs)) \\ \texttt{LSwapFirstTwo}(\mathsf{x}, \mathsf{v}) \\ (ok : \exists v_1, v_2, vs'.\ \ vs = v_1 : v_2 : vs' \ \star \mathsf{list}(x, v_2 : v_1 : vs') \star \mathsf{ret} = \texttt{null}) \\ (err : \mathsf{list}(x, vs) \star \ |vs| < 2 \ \star \mathsf{err} = \text{“}\texttt{List too short!}\text{”})\end{array}$$

observing that the success post-condition, given the used abstraction, has to not only state that the length of the list is not less than two, but also how the values are manipulated, and also that the error message is chosen for illustrative purposes.

However, note that the swapped values *are not featured in the function output*, but instead remain contained within the predicate. This indicates that a more abstract specification:

$$\begin{array}{c}(\mathsf{x} = x \star \mathsf{list}(x, n)) \\ \texttt{LSwapFirstTwo}(\mathsf{x}, \mathsf{v}) \\ (ok : \mathsf{list}(x, n) \star \ n \geq 2 \ \star \mathsf{ret} = \texttt{null})\ \ (err : \mathsf{list}(x, n) \star \ n < 2 \ \star \mathsf{err} = \text{“}\texttt{List too short!}\text{”})\end{array}$$

which only reveals the list length, might be EX-valid, and indeed it is. Any list we choose in the error post-model will have length less than two, and can then be used in the pre-model to reach the post-model. On the other hand, whichever list we choose in the success post-model will have length at least two, that is, its values will be of the form $v_1 : v_2 : vs$ and it will have some addresses, and then we can choose a list with the same addresses and values $v_2 : v_1 : vs$ in the pre-model and we will reach the post-model by executing the function.

Pointer-Reverse. Let us now examine the list-pointer-reverse function, $\texttt{LPRev}(\mathsf{x})$, which takes a list at address x and reverses it by reversing the direction of the next-pointers, returning the head of the reversed list. Given (O2) and the fact that the algorithm manipulates pointers and returns an address, but the actual values in the list are not exposed, we will try to use the address-only $\mathsf{list}(x, xs)$ predicate to specify this function as in the following OX triple, where $xs^\dagger$ denotes the reverse of the mathematical list xs:

$$\{\mathsf{x} = x \star \mathsf{list}(x, xs)\} \quad \texttt{LPRev}(\mathsf{x}) \quad \{\mathsf{list}(\mathsf{ret}, xs^\dagger)\}$$

which would seem to be UX-valid given our OX experience and previous examples, but is not. In particular, it has no information about the logical variable x, which exists only in the pre-condition. This is not an issue in OX reasoning, but in UX reasoning it would mean that there exists a logical environment that interprets the post-condition but not the pre-condition, and such a specification, by the definition, could never be UX-valid.

To understand which specific information about x is required, we first add the general $x \in \mathsf{Val}$, making the post-condition $\mathsf{list}(\mathsf{ret}, xs^\dagger) \star x \in \mathsf{Val}$, and then try to choose a post-model by picking values for ret, xs, and x. Note that, given the definition of $\mathsf{list}(x, xs)$, we cannot just pick any non-correlated values for ret and xs: in particular, either xs is an empty list and ret is $\texttt{null}$, or xs is non-empty and ret is its last element. This observation, in fact, reveals the information needed about x: either x is $\texttt{null}$ and xs is empty, or xs is non-empty and x is its first element. We capture this information using the $\mathsf{listHead}(x, xs)$ predicate:

$$\mathsf{listHead}(x, xs) \triangleq (xs = [\,] \star x = \texttt{null}) \lor (\exists xs'.\, xs = x : xs')$$

and arrive at the desired EX specification of the list-pointer-reverse algorithm:

$$(\mathsf{x} = x \star \mathsf{list}(x, xs)) \quad \texttt{LPRev}(\mathsf{x}) \quad (\mathsf{list}(\mathsf{ret}, xs^\dagger) \star \mathsf{listHead}(x, xs))$$

Let us make sure that this specification is UX-valid. If we pick a post-model with $xs = [\,]$, then $x = \mathsf{ret} = \texttt{null}$ and the pre-model with the same x and xs will work, as the list holds no values. For a post-model with non-empty xs, x must equal the head of xs, ret must equal the tail of xs, and we also have to pick some arbitrary values vs, with $|vs| = |xs|$. Then, given the described behaviour of the algorithm, we know that this post-model is reachable from a pre-model which has the list at x with addresses xs and values $vs^\dagger$.

Free. Next, we take a look at the $\mathsf{LFree}(\mathsf{x})$ function, which frees a given list at address x. Its OX specification is $\{\mathsf{x} = x \star \mathsf{list}(\mathsf{x})\}\ \texttt{LFree}(\mathsf{x})\ \{\mathsf{ret} = \texttt{null}\}$, but it does not transfer to UX contexts because no resource from the pre-condition can be forgotten in the post-condition as that would break the UX frame property [28]. Instead, we have to keep track of the addresses to be freed, which we can do using the $\mathsf{list}(x, xs)$ predicate (or $\mathsf{list}(x, xs, vs)$), and we also have to explicitly state in the post-condition that these addresses have been freed:

$$(\mathsf{x} = x \star \mathsf{list}(x, xs)) \quad \texttt{LFree}(\mathsf{x}) \quad (\mathsf{freed}(xs) \star \mathsf{listHead}(x, xs) \star \mathsf{ret} = \texttt{null})$$

using the $\mathsf{freed}(xs)$ predicate, which is defined as follows:

$$\mathsf{freed}(xs) \triangleq (xs = [\,]) \lor (\exists x, xs'.\, xs = x : xs' \star x \mapsto \varnothing, \varnothing \star \mathsf{freed}(xs'))$$

Client Programs and Specification Composition. We discuss the usability of ESL specifications in general and abstraction in particular in the context of client programs that call multiple library functions. Consider the following (slightly contrived) client program, which takes a list and: pointer-reverses it if its length is between 5 and 10; frees it and then throws an error if its length is smaller than 5; and does not terminate otherwise:

```
LClient(x) {
  l := LLen(x);
  if (l < 5)
    { r := LFree(x); error("List too short!") } else
    { if (l > 10) { while (true) { skip } } else { r := LPRev(l) } };
  return r
}
```

Our first goal is to understand which is the most abstract list predicate that could be used for reasoning about this client, since we want to minimise the amount of details we need to carry along in the proof, noting that the least abstract one, $\mathsf{list}(x, xs, vs)$, will always work. Observe that, importantly, only specifications expressed at the same abstraction level are composable with each other, because they must be composed using equivalence. We explore this in more detail in the subsequent formal discussion (see, in particular, observation (O5)).

When it comes to `LClient(x)`, for list-length, we need information about the list length, meaning that we can use either $\mathsf{list}(x, n)$, $\mathsf{list}(x, xs)$, or $\mathsf{list}(x, vs)$, but not $\mathsf{list}(x)$. For list-free, we must have information about the addresses, meaning that $\mathsf{list}(x, n)$ and $\mathsf{list}(x, vs)$ will not work, leaving us with $\mathsf{list}(x, xs)$, which is also usable for list-pointer-reverse. Therefore, we can write the specification of this client using the $\mathsf{list}(x, xs)$ predicate, as follows:

$$\begin{array}{c}(\mathsf{x} = x \star \mathsf{list}(x, xs))\\ \texttt{LClient(x)}\\ (ok : 5 \leq |xs| \leq 10 \star \mathsf{list}(\mathsf{ret}, xs^{\dagger}) \star \mathsf{listHead}(x, xs))\\ (err : |xs| < 5 \star \mathsf{freed}(xs) \star \mathsf{listHead}(x, xs) \star \mathsf{err} = \text{“}\texttt{List too short!}\text{”})\end{array}$$

In general, however, it is sufficient for a client to call one function that works with addresses and another that works with values for the only applicable predicate to be $\mathsf{list}(x, xs, vs)$, which is still abstract in the sense that it allows for unbounded reasoning about lists, but does not hide any of its internal information. This leads us to the following observation:

(O3) specifications that use predicates which hide data-structure information, albeit provable, may have limited use in UX client reasoning.

As a final remark on abstraction, note that we have only considered predicates that expose the data-structure sub-parts (for lists, these sub-parts are values vs and addresses xs) either entirely or not at all. It would be also possible to expose *some* of this structure for some of the algorithms, but because of (O3), specifications using such abstractions are only composable with specifications exposing the same partial structure, and hence likely to be of limited use.

Non-termination. We conclude our discussion on specifications with two remarks on EX reasoning about non-terminating behaviour. First, consider the non-terminating branch of the `LClient` function, which is triggered when $|xs| > 10$. Observe that this branch is implicit in the client specification, in that it is subsumed by the success post-condition (since $\models P \vee (|xs| > 10 \star \mathsf{False}) \Leftrightarrow P$). However, to demonstrate that it exists, we can constrain the pre-condition appropriately to prove the specification $(\mathsf{x} = x \star \mathsf{list}(x, xs) \star |xs| > 10)$ `LClient(x)` (False). This implicit loss of non-terminating branches can be characterised informally as follows:

(O4) 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 $|xs| \in \mathbb{N}$ and the post-conditions cover the cases where $|xs| \leq 10$, leaving the gap when $|xs| > 10$, for which we provably have client non-termination.

Second, we observe that, in contrast to terminating behaviour, for non-terminating behaviour EX is as expressive as OX; that is, the EX triple (P) C (False) is equivalent to the OX triple $\{P\}$ C $\{\mathsf{False}\}$ as the UX triple $[P]$ C $[\mathsf{False}]$ is vacuously true. This is not to say that all non-terminating behaviour can be captured by ESL specifications. For example, as in OX, if the code branches on a value that does not come 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 (O4) 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:

$(\ \mathsf{x} = 0\ )$ x := `nondet`; `if` (x > 42) { `while` (true) {`skip`} } `else` { x := 0 } $(\ \mathsf{x} = 0\ )$

5.3 More ESL Proofs: Iterative list-length

In §2, we have shown a proof sketch for a recursive implementation of the list-length algorithm, demonstrating 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. We highlight again the UX-specific issue that we raised and that is related to predicate folding, which can be formulated generally as follows:

(O5) if the code accesses data-structure information that the used predicate hides, then that predicate might not be foldable in a UX-proof in all of the places in which it would be foldable in the corresponding OX-proof.

Here, we show how to write ESL proofs for looping code, using as example an iterative implementation of the list-length algorithm. Proofs for the majority of the other algorithms mentioned in §5.2 can be found in [24]; the rest are similar.

Iterative list-length in ESL: Proof Sketch. In Figure 4, we give an iterative implementation of the list-length algorithm and show that it satisfies the same ESL specification as its recursive counterpart, $(\mathsf{x} = x \star \mathsf{list}(x, n))$ `LLen`(x) $(ok : \mathsf{list}(x, n) \star \mathsf{ret} = n)$. Since there is no recursion, we elide the (trivial) measure. To state the loop variant, we use the list-segment predicate, defined as follows:

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

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

$$P_i \triangleq \exists j.\ \mathsf{lseg}(x, \mathsf{x}, i) \star \mathsf{list}(\mathsf{x}, j) \star n = i + j \star \mathsf{r} = i$$

Note that we could have chosen to elide i from the body of P_i in this simple example, but since this is not necessarily possible or evident in general as well as for instructive purposes, we chose to keep it in the proof. Note how, on exiting the loop, the negation of the loop condition collapses the existentials i and j. This allows us to obtain the given internal post-condition, from which we then easily move to the desired external post-condition. For this proof, we also use three equivalence lemmas, which state that a non-empty list segment can be separated into its last element and the rest, that the length of an empty list equals zero, and that a null-terminated list-segment is a list.

$\Gamma \vdash$ ($\mathsf{x} = x \star \mathsf{list}(x, n)$)
LLen(x) {
($\mathsf{x} = x \star \mathsf{list}(x, n) \star \mathsf{r} = \mathtt{null}$)
r := 0
($\mathsf{x} = x \star \mathsf{list}(x, n) \star \mathsf{r} = 0$)
(P_0)
while (x ≠ null) {
($P_i \star \mathsf{x} \neq \mathtt{null}$)
($\exists j, v, x'.\ \mathsf{lseg}(x, \mathsf{x}, i) \star \mathsf{x} \mapsto v, x' \star \mathsf{list}(x', j-1) \star n = i + j \star \mathsf{r} = i$)
x := [x + 1];
($\exists j, v, x'.\ \mathsf{lseg}(x, x', i) \star x' \mapsto v, \mathsf{x} \star \mathsf{list}(\mathsf{x}, j) \star n = i + (j+1) \star \mathsf{r} = i$)
// equivalence: $\models \mathsf{lseg}(x, y, n+1) \Leftrightarrow \exists x', v.\ \mathsf{lseg}(x, x', n) \star x' \mapsto v, y]$
($\exists j.\ \mathsf{lseg}(x, \mathsf{x}, i+1) \star \mathsf{list}(\mathsf{x}, j) \star n = (i+1) + j \star \mathsf{r} = i$)
r := r + 1
(P_{i+1})
}
($\mathsf{x} = \mathtt{null} \star \exists i.\ P_i$)
($\exists i, j.\ \mathsf{lseg}(x, \mathsf{x}, i) \star \mathsf{list}(\mathsf{x}, j) \star n = i + j \star \mathsf{r} = i \star \mathsf{x} = \mathtt{null}$)
($\mathsf{lseg}(x, \mathtt{null}, n) \star \mathsf{r} = n \star \mathsf{x} = \mathtt{null}$) // equivalence: $\models \mathsf{list}(\mathtt{null}, j) \Leftrightarrow j = 0$
($\mathsf{list}(x, n) \star \mathsf{r} = n \star \mathsf{x} = \mathtt{null}$) // equivalence: $\models \mathsf{lseg}(x, \mathtt{null}, n) \Leftrightarrow \mathsf{list}(x, n)$
return r
($\mathsf{list}(x, n) \star \mathsf{r} = n \star \mathsf{x} = \mathtt{null} \star \mathsf{ret} = \mathsf{r}$)
($\exists x_q, r_q.\ \mathsf{list}(x, n) \star r_q = n \star x_q = \mathtt{null} \star \mathsf{ret} = r_q$)
($\mathsf{list}(x, n) \star \mathsf{ret} = n$)
}
($\mathsf{list}(x, n) \star \mathsf{ret} = n$)

Figure 4 ESL proof sketch: iterative list-length.

5.4 Beyond List Examples: Binary Search Trees

While list algorithms illustrate many aspects of exact reasoning, it is also important to understand how ESL specification and verification works with other data structures. For this reason, we discuss two algorithms operating over binary search trees (BSTs) that are intended to represent sets of natural numbers. We use two abstractions for BSTs, one in which only their values are considered as a mathematical set:

$$\mathsf{BST}(x, K) \triangleq (x = \mathtt{null} \star K = \emptyset) \vee (\exists k, l, r, K_l, K_r.\ x \mapsto k, l, r \star \mathsf{BST}(r, K_r) \star \mathsf{BST}(l, K_l) \star K = K_l \uplus \{k\} \uplus K_r \star K_l < k \star k < K_r)$$

and another that fully exposes the BST structure:

$$\mathsf{BST}(x, \tau) \triangleq (x = \mathtt{null} \star \tau = \tau_\emptyset) \vee (\exists k, l, r, \tau_l, \tau_r.\ E \mapsto k, l, r \star \mathsf{BST}(r, \tau_r) \star \mathsf{BST}(l, \tau_l) \star \tau = ((x, k), \tau_l, \tau_r) \star \tau_l < k \star k < \tau_r)$$

where τ is a mathematical tree, that is, an algebraic data type with two constructors representing, respectively, an empty tree and a root node with two child trees: $\tau \in \mathsf{Tree} \triangleq \tau_\emptyset \mid ((x, n), \tau_l, \tau_r)$, where the notation (x, n) represents a BST node with address x and value n. Note the overloaded $<$ notation, where one of the operands can be a set or a tree, which carry the intuitive meaning.

BST algorithms. We first consider the BST-find-minimum algorithm, `BSTFindMin(x)`, which takes a tree with root at x, does not modify it, and returns its minimum element or throws an empty-tree error. Since that algorithm operates only on the values in the tree, we are able to state its ESL specification using the $\mathsf{BST}(x, K)$ predicate as follows:

$$(\mathsf{x} = x \star \mathsf{BST}(x, K))\ \texttt{BSTFindMin(x)}\ \begin{array}{l}(ok\colon\ x \neq \texttt{null} \star \mathsf{BST}(x, K) \star \mathsf{ret} = \min(K))\\ (err\colon\ x = \texttt{null} \star \mathsf{BST}(x, K) \star \mathsf{err} = \text{“}\texttt{Empty tree!}\text{”})\end{array}$$

We have also considered the BST-insert algorithm, `BSTInsert(x,v)`, which takes a tree with root at x and inserts a new node with value v into it as a leaf if v is not already in the tree, or leaves the tree unmodified if it is. As this algorithm interacts both with values and addresses in the tree, the appropriate abstraction for it is $\mathsf{BST}(x, \tau)$, and its ESL specification is:

$$\begin{array}{c}(\mathsf{x} = x \star \mathsf{v} = v \star \mathsf{BST}(x, \tau))\\ \texttt{BSTInsert(x,v)}\\ (\exists x'.\ \mathsf{BST}(\mathsf{ret}, \mathsf{BSTInsert}(\tau, (x', v))) \star \mathsf{BSTRoot}(x, \tau))\end{array}$$

where $\mathsf{BSTInsert}(\tau, \nu)$ is the mathematical algorithm that inserts the node ν into the tree τ:

$$\begin{array}{ll}\mathsf{BSTInsert}(\tau_\emptyset, (x', v)) \triangleq & \mathsf{BSTInsert}(((x, k), \tau_l, \tau_r), (x', v)) \triangleq\\ \quad ((x', v), \tau_\emptyset, \tau_\emptyset) & \quad \mathsf{if}\ v < k\ \mathsf{then}\ ((x, k), \mathsf{BSTInsert}(\tau_l, (x', v)), \tau_r)\\ & \quad \mathsf{else\ if}\ k < v\ \mathsf{then}\ ((x, k), \tau_l, \mathsf{BSTInsert}(\tau_r, (x', v)))\\ & \quad \mathsf{else}((x, k), \tau_l, \tau_r)\end{array}$$

and the predicate $\mathsf{BSTRoot}(x, \tau)$ is defined analogously to $\mathsf{listHead}(x, xs)$:

$$\mathsf{BSTRoot}(x, \tau) \triangleq (\tau = \tau_\emptyset \star x = \texttt{null}) \vee (\exists k, \tau_l, \tau_r.\ \tau = ((x, k), \tau_l, \tau_r))$$

This example shows how in EX verification, just as in OX verification, we end up relating an imperative heap-manipulating algorithm to its mathematical/functional counterpart (cf. Appel [3] for a recent reiteration of this idea). The additional work required is that EX mathematical models must be more detailed: we are, yet again, not allowed to lose information. In particular, in OX verification we could relate `BSTInsert(x,v)` to mathematical sets, but in EX verification we must relate our imperative implementation to tree models, including both values and pointers. Moreover, our mathematical model of the algorithm, $\mathsf{BSTInsert}(\tau, (x', v))$, must insert elements in the same way as the imperative implementation, that is, in this case at the leaves of the tree. The proofs for both algorithms are given in [24].

6 Related Work

In the previous sections, we have placed ESL in the context of related work on OX and UX logics and associated tools. Here, we discuss formalisms capable of reasoning both about program correctness and program incorrectness, as well as existing approaches to the use of function specifications (summaries) and abstraction in symbolic execution.

Program Logics for Both Correctness and Incorrectness. Developed in parallel but independently of ESL, Outcome Logic (OL) [33], much like ESL, brings together reasoning about correctness and incorrectness into one logic. Both OL and ESL rely on the traditional meaning of correctness, but OL introduces a new approach to incorrectness, based on reachability of sets of states. It has not yet been shown that this approach has the same bug-finding potential as that of ISL: in particular, bi-abduction has not yet been demonstrated to be compatible with OL. In addition, the OL work, in contrast to ESL, does not discuss function compositionality or the interaction between abstraction, reachability, and incorrectness.

LCL_A [4, 5] is a non-function-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 $\vdash_A [P]\ \mathsf{C}\ [Q]$ 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.

Compositional Symbolic Execution. There exists a substantial body of work on symbolic execution with function summaries (e.g. [1, 15–17, 22, 32]), which is primarily based on first-order logic. We highlight the work of Godefroid et al., which initially used exact summaries of bounded program behaviour to drive the compositional dynamic test generation of SMART [15], and later distinguished between may (OX) and must (UX) summaries, leveraging the interaction between them to design the SMASH tool for compositional property checking and test generation [16]. SMASH, however, is limited in its ability to reason about heap-manipulating programs because, for example, it lacks support for pointer arithmetic. Nevertheless, it shows that interactions between OX and UX summaries can be exploited for automation, which is an important consideration for any automation of ESL. For example, SMASH is able to use not-may summaries (which amount to non-reachability) when constructing must-summaries (which amount to reachability), using the former to restrict the latter. 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.

7 Conclusions

We have introduced ESL, a program logic for exact reasoning about heap-manipulating programs. ESL specifications provide a sweet spot between verification and true bug-finding: as SL specifications, they capture all terminating behaviour, and, as ISL specifications, they describe only results and errors that are reachable. ESL specifications are therefore compatible with tools that target OX verification, such as VeriFast [19] and Iris [20], tools that target UX true bug-finding, such as Pulse [26, 28], and tools capable of targeting both, such as Gillian [10, 23]. ESL supports reasoning about mutually recursive functions and comes with a soundness result that immediately transfers to SL and ISL, thus demonstrating, for the first time, scalable functional compositionality for UX logics.

We have verified exact specifications for a number of illustrative examples, showing that ESL can reason about data-structure libraries, language errors, mutual recursion, and non-termination. In doing so, we emphasise the distinction between the often-conflated concepts of abstraction and over-approximation. We have demonstrated that abstract predicates can be soundly used in EX and UX reasoning, albeit not as freely as in OX reasoning.

We believe that ESL reasoning, in its intended context of semi-automatic verification of functional correctness properties, is useful for the verification of self-contained, critical code that underpins a larger codebase. To demonstrate this, we will in the future incorporate UX and EX verification inside Gillian [10, 23], which already has support for function compositionality and semi-automatic predicate management as part of its OX verification.

References

1. Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. doi:10.1007/978-3-540-78800-3_28.
2. Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1), 2009. doi:10.1007/s10009-008-0090-1.
3. Andrew W. Appel. Coq's vibrant ecosystem for verification engineering. In *Conference on Certified Programs and Proofs (CPP)*, 2022. doi:10.1145/3497775.3503951.
4. Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A logic for locally complete abstract interpretations. In *Symposium on Logic in Computer Science (LICS)*, 2021. doi:10.1109/LICS52264.2021.9470608.
5. Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A correctness and incorrectness program logic. *Journal of the ACM*, 70(2), 2023. doi:10.1145/3582267.
6. Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages*, 4(ICFP), 2020. doi:10.1145/3408998.
7. Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *European Symposium on Programming (ESOP)*, 2016. doi:10.1007/978-3-662-49498-1_8.
8. Edsko de Vries and Vasileios Koutavas. Reverse Hoare logic. In *Software Engineering and Formal Methods (SEFM)*, 2011. doi:10.1007/978-3-642-24690-6_12.
9. Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19, 1967.
10. José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part I: A multi-language platform for symbolic execution. In *Programming Language Design and Implementation (PLDI)*, 2020. doi:10.1145/3385412.3386014.
11. José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *Principles and Practice of Declarative Programming (PPDP)*, 2018. doi:10.1145/3236950.3236956.
12. José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: JavaScript verification toolchain. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158138.
13. José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290379.
14. Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103656.2103663.
15. Patrice Godefroid. Compositional dynamic test generation. In *Principles of Programming Languages (POPL)*, 2007. doi:10.1145/1190216.1190226.
16. Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Principles of Programming Languages (POPL)*, 2010. doi:10.1145/1706299.1706307.
17. Benjamin Hillery, Eric Mercer, Neha Rungta, and Suzette Person. Exact heap summaries for symbolic execution. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2016. doi:10.1007/978-3-662-49122-5_10.
18. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)*, 12(10), 1969. doi:10.1145/363235.363259.
19. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium (NFM)*, 2011. doi:10.1007/978-3-642-20398-5_4.

20 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2676726.2676980.

21 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), 2022. doi:10.1145/3527325.

22 Yude Lin, Tim Miller, and Harald Sondergaard. Compositional symbolic execution using fine-grained summaries. In *Australasian Software Engineering Conference*, 2015. doi:10.1109/ASWEC.2015.32.

23 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: Real-world verification for JavaScript and C. In *Computer Aided Verification (CAV)*, 2021. doi:10.1007/978-3-030-81688-9_38.

24 Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. Exact separation logic (extended version), 2023. arXiv:2208.07200.

25 Toby Murray, Pengbo Yan, and Gidon Ernst. Incremental vulnerability detection with insecurity separation logic, 2021. arXiv:2107.05225.

26 Peter W. O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2019. doi:10.1145/3371078.

27 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, 2001. doi:10.1007/3-540-44802-0_1.

28 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification (CAV)*, 2020. doi:10.1007/978-3-030-53291-8_14.

29 Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Concurrent incorrectness separation logic. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022. doi:10.1145/3498695.

30 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002. doi:10.1109/LICS.2002.1029817.

31 Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction, Chapter 10*. MIT Press, 1993.

32 Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Principles of Programming Languages (POPL)*, 2008. doi:10.1145/1328438.1328467.

33 Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 2023. doi:10.1145/3586045.