Staged Specification Logic for Verifying Higher-Order Imperative Programs

Darius Foo(✉)[0000−0002−3279−5827], Yahui Song[0000−0002−9760−5895], and Wei-Ngan Chin[0000−0002−9660−5682]

School of Computing, National University of Singapore, Singapore
{dariusf,yahuis,chinwn}@comp.nus.edu.sg

Abstract. Higher-order functions and imperative states are language features supported by many mainstream languages. Their combination is expressive and useful, but complicates specification and reasoning, due to the use of yet-to-be-instantiated function parameters. One inherent limitation of existing specification mechanisms is its reliance on *only two stages* : an initial stage to denote the precondition at the start of the method and a final stage to capture the postcondition. Such two-stage specifications force *abstract properties* to be imposed on unknown function parameters, leading to less precise specifications for higher-order methods. To overcome this limitation, we introduce a novel extension to Hoare logic that supports *multiple stages* for a call-by-value higher-order language with ML-like local references. Multiple stages allow the behavior of unknown function-type parameters to be captured abstractly as uninterpreted relations; and can also model the repetitive behavior of each recursion as a separate stage. In this paper, we define our staged logic with its semantics, prove its soundness and develop a new automated higher-order verifier, called Heifer, for a core ML-like language.

1 Introduction

Programs written in modern languages today are rife with higher-order functions [3, 35], but specifying and verifying them remains challenging, especially if they contain imperative effects. Consider the *foldr* function from OCaml. Here is a good specification for it in Iris [17], a state-of-the-art framework for higher-order concurrent separation logic that is built using Coq proof assistant.

$$\forall\, P, Inv, f, xs, l.\left\{\begin{array}{l}(\forall\, x, a', ys.\, \{P\ x * Inv\ ys\ a'\}\ f(x, a')\ \{r.\ Inv\ (x{::}ys)\ r\})\\ *\, isList\ l\ xs * all\ P\ xs * Inv\ []\ a\end{array}\right\}$$
$$foldr\ f\ a\ l$$
$$\{r.\ isList\ l\ xs * Inv\ xs\ r\}$$

While this specification is conventional in weakest-precondition calculi like Iris, one might argue that that this specification is not the best possible specification for *foldr*, since it requires two *abstract properties* *Inv* and *P* to summarize the behaviour of *f*. Moreover, the input list *l* is also immutable, through the same *isList* predicate in both its pre- and postcondition. (If mutation of list is allowed, a more complex *Inv* with an extra mutated list parameter is required.)

These abstract properties must be correspondingly instantiated for each instance of f, but unfortunately some usage scenarios (to be highlighted later in Sec 2.2) of *foldr* cannot be captured by this particular pre/post specification of Iris, despite how well-designed it was. Thus, the conventional pre/post approach to specifying higher-order functions currently suffers from possible *loss in precision* in its specifications since the presence of these abstract properties implicitly *strengthens* the preconditions for higher-order imperative methods.

This paper proposes a new logic, *Higher-Order Staged Specification Logic* (*HSSL*), for specifying and verifying higher-order imperative methods. It is designed for automated verification via SMT and uses separation logic as its core stateful logic, aiming at more precise specifications of heap-based changes. While we have adopted separation logic to support heap-based mutations, *HSSL* may also be used with other base logics, such as those using dynamic frames [25]. We next provide an overview of our methodology by examples before providing formal details and an experimental evaluation of our proposal.

2 Illustrative Examples

We provide three examples to highlight the key features of our methodology.

2.1 A Simple Example

We introduce the specification logic using a simple example (Fig. 1), to highlight a key challenge we hope to solve, namely how should we specify the behavior of *hello* without pre-committing to some abstract property on f? To do that, we can model f using an *uninterpreted relation*. We use uninterpreted relation rather than a function here in order to model both over-approximation and possible side-effect. Since f is effectful and may modify arbitrary state, including the references x and y, a modular specification of *hello* must express the ordering of the call to f with respect to the other statements in it so that the caller of *hello* may reason precisely about its effects. Therefore, a first approximation is the following specification. We adopt standard separation logic pre/post assertions and extend them with *sequential composition* and *uninterpreted relations*. A final parameter (named as *res* here) is added to denote the result of each staged specification's relation (*hello* here), a convention we follow henceforth.

```
let hello f x y =
  x := !x + 1;
  let r = f y in
  let r2 = !x + r in
  y := r2;
  r2
```

Fig. 1: A Simple Example

$hello(f, x, y, res) =$

$\exists\, a \cdot \mathbf{req}\ x \mapsto a;$	// Stage 1: requiring x be pre-allocated
$\mathbf{ens}[\_]\ x \mapsto a{+}1;$	// Stage 2: ensuring x is updated
$\exists\, r \cdot f(y, r);$	// Stage 3: unknown higher-order f call
$\exists\, b \cdot \mathbf{req}\ x \mapsto b * y \mapsto \_;$	// Stage 4: requiring x, y be pre-allocated
$\mathbf{ens}[res]\ x \mapsto b * y \mapsto res \wedge res{=}b{+}r$	// Stage 5: y is updated, and x is unchanged

We can summarize the imperative behavior of *hello* before the call to f with a read from x, followed by a write to x, as captured by Stages 1-2. The same applies to the portion after the call to f (lines 4-6), but here we only consider the scenario when x and y are disjoint [1]. Stages 4 and 5 state that memory location x is being read while y will be correspondingly updated.

The ordering of the unknown f call with respect to the parts before and after does matter, so the call can be seen as *stratifying* the temporal behavior of the function into *stages.* Should a specification for f become known, usually at a call site, its *instantiation* may lead to a staged formula with only **req**/**ens** stages; which can always be *compacted* into a single **req**/**ens** pair. We detail a *normalization* procedure to do this in Section 3.2.

As mentioned before, f can modify x despite not having direct access to it via an argument, as it could capture x from the environment of the caller of *hello*. To model this, we make worst-case assumptions on the footprints of unknown functions, resulting in the precondition $x \mapsto b$ in stage 4.

2.2 Pre/Post vs Staged Specifications via *foldr*

We now specify *foldr* and compare it with the Iris specification from Section 1.

```
let rec foldr f a l =
  match l with
  | [] => a
  | h :: t =>
    f h (foldr f a t)
```

$$\begin{aligned} & \mathit{foldr}(f, a, l, rr) = \\ & \quad \mathbf{ens}[rr]\ l{=}[] \wedge rr{=}a \\ & \vee\ \exists\, x, r, l_1 \cdot\ \mathbf{ens}[\_]\ l{=}x{::}l_1; \\ & \qquad \mathit{foldr}(f, a, l_1, r); f(x, r, rr) \end{aligned}$$

We model *foldr* as a recursive predicate whose body is a staged formula. The top-level disjunction represents the two possible paths that result from pattern matching. In the base case, when l is the empty list, and the result of *foldr* is a. In the recursive case, when l is nonempty, the specification expresses that the behavior of *foldr* is given by a recursive call to *foldr* on the tail of l to produce a result r, followed by a call to f with r to produce a value for res. Crucially, we are able to represent the call to the unknown function f directly in the specification, without being forced to impose a stronger precondition on f.

foldr's specification's is actually very precise, to the point of mirroring the *foldr* program. Nevertheless, abstraction may readily be recovered by proving that this predicate entails a weaker formula, and a convenient point for this would be when the unknown function-typed parameter is instantiated at each of *foldr*'s call sites; we discuss an example of this shortly. The point of specifying *foldr* this way is that the precision of stages enables us not to have to commit to an abstraction prematurely. We should, of course, summarize as early as is appropriate to keep our proving process tractable.

[1] For simplicity, the intersection of specifications $\wedge_{sp}$ that arises from disjoint preconditions is omitted (with some loss in precision) in the main paper, but its core mechanism is briefly described in the appendix [13].

Recursive staged formulae are needed mainly to specify higher-order functions with unknown function-typed parameters. Otherwise, our preference is to apply summarization to obtain non-recursive staged formulae whenever unknown function-type parameters have been suitably instantiated. Under this scenario, we may still use recursive pure predicates or recursive shape predicates in order to obtain best possible modular specifications for our program code.

Now, we show how the staged specification for *foldr* can be used by proving that we can sum a list by folding it. *sum* can be specified in a similar way to *foldr*, but since this is a pure function that can be additionally checked for termination, we can automatically convert it into a *pure predicate* (without any stages or imperative side effects) to be used in (the pure fragment of) our specification logic. Termination of pure predicates is required for them to be safely used in specifications. (Techniques to check for purity and termination are well-known and thus omitted.) Also, each pure predicate may be used as either a staged predicate or a pure predicate. In case a pure predicate $p(v^*, res)$ is used as a staged predicate; its staged definition is:

$$p(v^*, res) = \ \mathbf{req}\ emp{\wedge}pre(v^*); \mathbf{ens}[\_]\ emp{\wedge}p(v^*, res)$$

where $pre(v^*)$ denotes the precondition to guarantee termination and avoids exceptions. Note that $p(v^*, res)$ is overloaded to be used as either a staged predicate or a pure predicate. This is unambiguous from the context of its use.

```
let rec sum li =
  match li with
  | [] -> 0
  | x :: xs -> x + sum xs
```

$$\begin{aligned} &sum(li, res) = \\ &\quad l{=}[]{\wedge}res{=}0 \\ &\vee\ \exists\, r, l_1 \cdot l{=}x{::}l_1{\wedge}sum(l_1, r){\wedge}res{=}x{+}r \end{aligned}$$

We can now re-summarize an imperative use of *foldr* with the help of *sum*.

```
let foldr_sum_state x xs init
```
$foldr\_sum\_state(x, xs, init, res) =$
$\quad \exists\, i, r \cdot \mathbf{req}\ x \mapsto i; \mathbf{ens}[res]\ x \mapsto i{+}r{\wedge}res{=}r{+}init{\wedge}sum(xs, r)$
```
= let g c t = x := !x + c; c + t in foldr g xs init
```

This summarization gives rise to the following entailment:

$$\begin{aligned} &\forall\, m, xs, init, res. \, foldr(g, xs, init, res) \\ \sqsubseteq\ &\exists\, i, r \cdot \mathbf{req}\ x \mapsto i; \mathbf{ens}[res]\ x \mapsto i{+}r{\wedge}res{=}r{+}init{\wedge}sum(xs, r) \end{aligned}$$

We have implemented a proof system for subsumption (denoted by $\sqsubseteq$) between staged formulae in our verifier, called Heifer. This particular entailment can be proved automatically by induction on *xs*. While Iris's earlier pre/post specification for *foldr* can handle this example through a suitable instantiation of (*Inv* _ _), it is unable to handle the following three other call instances.

```
let foldr_ex1 l = foldr (fun x r -> let v = !x
                                    in x := v+1; v+r) l 0
let foldr_ex2 l = foldr (fun x r -> assert(x+r>=0);x+r) l 0
let foldr_ex3 l = foldr (fun x r -> if x>=0 then x+r
                                    else raise Exc()) l 0
```

The first example cannot be handled since Iris's current specification for *foldr* expects its input list l to be immutable. The second example fails since the precondition required cannot be expressed using just the abstract property $(P\ x)$. The last example fails because the abstract property $(\mathit{Inv}\ (x :: ys)\ r)$ used in the postcondition of f expects its method calls to return normally. In contrast, using our approach via staged specification, we can re-summarize the above three call instances to use the following subsumed specifications.

$$\begin{array}{l}
\mathit{foldr\_ex1}(l,res) \sqsubseteq \exists\, xs \cdot \mathbf{req}\ \ \mathit{List}(l,xs)\ ;\ \exists\, ys \cdot \\
\qquad\qquad\qquad\qquad \mathbf{ens}[res]\ \mathit{List}(l,ys) \wedge \mathit{mapinc}(xs,ys) \wedge \mathit{sum}(xs,res) \\
\mathit{foldr\_ex2}(l,res) \sqsubseteq \mathbf{req}\ \ \mathit{allSPos}(l)\ ;\ \mathbf{ens}[res]\ \mathit{sum}(l,res) \\
\mathit{foldr\_ex3}(l,res) \sqsubseteq \mathbf{ens}[res]\ \mathit{allPos}(l) \wedge \mathit{sum}(l,res)\ \vee\ (\mathbf{ens}[\_]\ \neg \mathit{allPos}(l); \mathit{Exc}())
\end{array}$$

Note that the first example utilizes a recursive spatial $\mathit{List}(l,xs)$ predicate, while the last example used $\mathit{Exc}()$ as a relation to model exception as a stage in our specification. The three pure predicates and one spatial predicate used in the above can be formally defined, as shown below.

$$\begin{array}{lcl}
\mathit{mapinc}(xs,ys) & = & (xs{=}[] \wedge ys{=}[]) \vee (\exists\, x, xs_1, ys_1 \cdot xs{=}x{::}xs_1 \wedge ys{=}(x{+}1){::}ys_1 \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{mapinc}(xs_1, ys_1)) \\
\mathit{allPos}(l) & = & (l{=}[]) \vee (\exists\, x, l_1 \cdot l{=}x{::}l_1 \wedge \mathit{allPos}(l_1) \wedge x{\geq}0) \\
\mathit{allSPos}(l) & = & (l{=}[]) \vee (\exists\, x, r, l_1 \cdot l{=}x{::}l_1 \wedge \mathit{allSPos}(l_1) \wedge \mathit{sum}(l,r) \wedge r{\geq}0) \\
\mathit{List}(l,rs) & = & (\mathit{emp} \wedge l{=}[]) \vee (\exists\, x, rs_1, l_1 \cdot x \mapsto r * \mathit{List}(l_1,rs_1) \wedge l{=}x{::}l_1 \wedge rs{=}r{::}rs_1)
\end{array}$$

We emphasize that our proposal for staged logics is strictly more *expressive* than traditional two-stage pre/post specifications, since the latter can be viewed as an instance of staged logics. As an example, the earlier two-stage specification for *foldr* can be modelled non-recursively in our staged logics as:

$$\begin{array}{l}
\mathit{foldr}(f,a,l,res) = \\
\quad \exists\, P, \mathit{Inv}, xs \cdot \mathbf{req}\ \mathit{List}(l,xs) * \mathit{Inv}([],a) \wedge \mathit{all}(P,xs) \\
\qquad \wedge f(x,a',r) \sqsubseteq (\exists\, ys \cdot \mathbf{req}\ \mathit{Inv}(ys,a') \wedge P(x); \mathbf{ens}[r]\ \mathit{Inv}(x{::}ys,r))\ ; \\
\quad \mathbf{ens}[res]\ \mathit{List}(l,xs) * \mathit{Inv}(xs,res)
\end{array}$$

2.3 Inferrable vs User-provided Specifications via *map*

Our methodology for higher-order functions is further explicated by the *map* method, shown in Fig. 2. Specifications typeset in `lavender` must be user-supplied, whereas those shown in °`red` (with the small circle) may be automated or inferred (using the rules of Section 4). Like *sum* before, *length* and *incrg* may be viewed as *ghost* functions, written only for their specifications to be used to describe behavior. These specifications are also routine and can be mechanically derived; we elide them here and provide them in the appendix [13]. The method

```
let rec length xs =
 $^\circ length(xs, res) = ...$
  match xs with
  | [] -> 0
  | x :: xs1 ->
    1 + length xs1

let rec incrg init li =
$^\circ incrg(init, li, res) = ...$
  match li with
  | [] -> []
  | x :: xs -> init ::
      incrg (init + 1) xs
```

```
let rec map f xs =
$^\circ map(f, xs, res) = ...$
  match xs with
  | [] -> []
  | x :: xs1 ->
    f x :: map f xs1

let map_incr xs x
$map\_incr(xs, x, r) =$
     $\exists i \cdot \mathbf{req}\ x \mapsto i;\ \exists m \cdot \mathbf{ens}[r]\ x \mapsto i{+}m$
      $\wedge\ length(xs, m) \wedge incrg(i{+}1, xs, r)$
= let f a = x := !x+1; !x
  in map f xs
```

Fig. 2: Implementation of *map_incr* with a Summarized Specification from *map*

map_incr describes the scenario we are interested in, where *the state of the closure affects the result of map*. Its specification states that the pointer x must have its value incremented by the length of xs. Moreover, the *contents* of the resulting list is captured by another pure function *incrg*, which builds a list of as many increasing values as there are elements in its input list.

These examples illustrate the methodology involved with staged specifications. They inherit the modular verification and biabduction-based[4] specification inference of separation logic, adding the ability to describe imperative behavior using function stages to the mix; biabduction then doubles as a means to normalize and compact stages. There is emphasis on the inference of specifications and proof automation, and proofs are built out of simple lemmas, which help summarize behavior and the shapes of data, and either remove recursion or move it into a pure ghost function where it is easier to comprehend.

In summary, staged logic for specifying imperative higher-order functions represents a fundamentally new approach that is *more general* and yet can be *more precise* than what is currently possible via state-of-the-art pre/post specification logics for imperative higher-order methods. Our main technical contributions to support this new approach include:

1. **Higher-Order Staged Specification Logic (*HSSL*):** we design a novel program logic to specify the behaviors of imperative higher-order methods and give its formal semantics.
2. **Biabduction-based Normalization:** we propose a normalization procedure for *HSSL* that serves two purposes: (i) it allows us to produce succinct staged formulae for programs automatically, and (ii) it helps structure entailment proof obligations, allowing them to be discharged via SMT.
3. **Entailment:** we develop a proof system to solve subsumption entailments between normalized *HSSL* formulae, prove its soundness, and implement an automated prover based on it.
4. **Evaluation:** we report on initial experimental results, and present various case studies highlighting *HSSL*'s capabilities.

3 Language and Specification Logic

We target a minimal OCaml-like imperative language with higher-order functions and state. The syntax is given in Fig. 3. Expressions are in ANF (A-normal form); sequencing and control over evaluation order may be achieved using let-bindings, which define immutable variables. Mutation may occur through heap-allocated *ref*s. Functions are defined by naming lambda expressions, which may be annotated with a specification Φ (covered below). For simplicity, they are always in tupled form and their calls are always fully applied. Pattern matching is encoded using recognizer functions (e.g., *is_cons*) and *if* statements. *assert* allows proofs of program properties to be carried out at arbitrary points.

$$
\begin{array}{lll}
(\mathit{Expressions}) & e ::= & v \mid x \mid \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 \mid f(x^*) \mid \mathit{ref}\ x \mid x_1 := x_2 \mid {!x} \mid x_1 :: x_2 \mid \\
 & & \mathit{assert}\ D \mid \mathit{if}\ x\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \\
(\mathit{Values}) & v ::= & c \mid \mathit{nil} \mid x_1{::}x_2 \mid \mathit{fun}\,(x^*)\,\Phi[r] \rightarrow e \\
(\mathit{Staged}) & \Phi ::= & E \mid \Phi_1 \vee \Phi_2 \mid \Phi_1\,;\Phi_2 \mid \exists\, x^* \cdot \Phi \\
(\mathit{Stage}) & E ::= & \mathbf{req}\ D \mid \mathbf{ens}[r]\ D \mid f(x^*, r) \qquad\qquad (\mathit{State}) \quad D ::= \sigma \wedge \pi \\
(\mathit{Heap}) & \sigma ::= & \mathit{emp} \mid x_1 \mapsto x_2 \mid \sigma_1 * \sigma_2 \\
(\mathit{Pure}) & \pi ::= & \mathit{true} \mid \pi_1{\vee}\pi_2 \mid \neg\pi \mid \exists\, x.\ \pi \mid t_1{=}t_2 \mid a_1{<}a_2 \mid \Phi_1{\sqsubseteq}\Phi_2 \\
(\mathit{A\text{-}Terms}) & a ::= & i \mid x \mid a_1 + a_2 \mid -a \\
(\mathit{Terms}) & t ::= & \mathit{nil} \mid t_1{::}t_2 \mid c \mid a \mid f \mid \lambda\,(x^*, r) \rightarrow \Phi
\end{array}
$$

$$c \in \mathbb{B} \cup \mathbb{Z} \cup \mathbf{unit} \qquad\qquad i \in \mathbb{Z} \qquad\qquad x, f, r \in \mathit{var}$$

Fig. 3: Syntax of the Core Language and Staged Logics

Program behavior is specified using *staged formulae* Φ, which are disjunctions and/or sequences of *stages* E. A stage is an assertion about program state *at a specific point*. Each stage takes one of three forms: a precondition $\mathbf{req}\ D$, a postcondition $\mathbf{ens}[r]\ D$ with a named result r, or a *function stage* $f(v^*, r)$, representing the specification of a (possibly-unknown) function call. For brevity, we use a context notation $\Phi[r]$ where r explictly identifies the final result of specification Φ. Program states D are described using separation logic formulae from the *symbolic heap* fragment [4], without recursive spatial predicates (for simplicity of presentation). Most values of the core language are as usual also terms of the (pure) logic; a notable exception is the lambda expression $\mathit{fun}\,(x^*)\,\Phi[r] \rightarrow e$, which occurs in the logic as $\lambda\,(x^*, r) \rightarrow \Phi[r]$, without its body. Subsumption assertions between two staged formulae (Sec 5) are denoted by $\Phi_1 \sqsubseteq \Phi_2$.

3.1 Semantics of Staged Formulae

From Triples to Stages Staged formulae generalize standard Hoare triples. The standard partial-correctness interpretation of the separation logic Hoare triple $\{\ P(v^*, x^*)\ \}\ e\ \{\ \exists y^* \cdot Q(v^*, x^*, y^*, \mathit{res})\ \}$ where v^* denote valid program variables and x^* denote specification variables (e.g., ghost variables) is that for all states st satisfying $P(v^*, x^*)$, given a reduction $e, st \rightsquigarrow^* v, st'$, if

$e, st \not\rightsquigarrow^* \mathit{fault}$, then st' satisfies $\exists y^* \cdot Q(v^*, x^*, y^*, \mathit{res})$. The staged equivalent is $\{\ \Phi\ \}\ e\ \{\ \Phi; \exists\, x^* \cdot \mathbf{req}\ P(v^*, x^*); \exists\, y^* \cdot \mathbf{ens}[\_]\ Q(v^*, x^*, y^*, \mathit{res})\ \}$. Apart from mentioning the *history* Φ, which remains unchanged, its meaning is identical. Consider, then, $\{\ \Phi\ \}\ e\ \{\ \Phi; \mathbf{req}\ P_1; \mathbf{ens}[\_]\ Q_1; \mathbf{req}\ P_2; \mathbf{ens}[\_]\ Q_2\ \}$ – an intuitive extension of the semantics of triples is that given $e, st \rightsquigarrow^* e_1, st_1$, where st_1 satisfies Q_1, the extended judgment holds if st_1 *further* satisfies P_2, and reduction from there, $e_1, st_1 \rightsquigarrow^* e_2, st_2$, results in a state st_2 that satisfies Q_2.

While heap formulae are satisfied by program states, staged formulae (like triples), are satisfied by traces which begin and end at particular states. Uninterpreted function stages further allow stages to describe the *intermediate states* of programs in specifications – a useful ability in the presence of unknown higher-order imperative functions, as we illustrate in Section 2 and the appendix [13]. To formalize all this, we give a semantics for staged formulae next.

Formal Semantics We first recall the standard semantics for separation logic formulae in Fig. 4, which provides a useful starting point.

$S, h \models \sigma \wedge \pi$	*iff*	$[\![\pi]\!]_S$ and $S, h \models \sigma$
$S, h \models \mathit{emp}$	*iff*	$\mathit{dom}(h) = \{\}$
$S, h \models x \mapsto y$	*iff*	$\mathit{dom}(h) = \{S(x)\}$ and $h(S(x)) = [\![y]\!]_S$
$S, h \models \sigma_1 * \sigma_2$	*iff*	$\exists h_1 h_2.\ h_1 \circ h_2 = h$ such that $S, h_1 \models \sigma_1$ and $S, h_2 \models \sigma_2$

Fig. 4: Semantics of Separation Logic Formulae

Let *var* be the set of program variables, *val* the set of primitive values, and $\mathit{loc} \subset \mathit{val}$ the set of heap locations; ℓ is a metavariable ranging over locations. The models are program states, comprising a *store* of variables S, a partial mapping from a finite set of variables to values $\mathit{var} \rightharpoonup \mathit{val}$, and the heap h, a partial mapping from locations to values $\mathit{loc} \rightharpoonup \mathit{val}$. $[\![\pi]\!]_S$ denotes the valuation of pure formula π under store S. $\mathit{dom}(h)$ denotes the domain of heap h. $h_1 \circ h_2 = h$ denotes disjoint union of heaps; if $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \{\}$, $h_1 \cup h_2 = h$. We write $h_1 \subseteq h_2$ to denote that h_1 is a subheap of h_2, i.e., $\exists\, h_3 \cdot h_1 \circ h_3 = h_2$. $s[x{:=}v]$ and $s[x{:\not=}]$ stand for store/heap updates and removal of keys.

We define the semantics of *HSSL* formulae in Fig. 5. Let $S, h \rightsquigarrow S_1, h_1, R \models \Phi$ denote the *models* relation, i.e., starting from the program state with store S and heap h, the formula Φ transforms the state into S_1, h_1, with an intermediate result R. R is either $\mathit{Norm}(r)$ for partial correctness, Err for precondition failure, or $\top$ for *possible* precondition failure in one of its execution paths.

When Φ is of the form $\mathbf{req}\ \sigma \wedge \pi$, the heap h is split into a heaplet h_1 satisfying $\sigma \wedge \pi$, which is consumed, and a frame h_2, which is left as the new heap. *Read-only heap assertions* $(\sigma \wedge \pi)@R$ under $\mathbf{req}$ check but do not change the heap.

When Φ is of the form $\mathbf{ens}[\_]\ \sigma \wedge \pi$, σ describes locations which are to be added to the current heap. The semantics allows some concrete heaplet h_1 that satisfies $\sigma \wedge \pi$ (containing new or updated locations) be (re-)added to heap h.

When Φ is a function stage $f(x^*, r)$, its semantics depends on the specification of f. A staged existential causes the store to be extended with a binding from x

$$
\begin{array}{ll}
S, h \rightsquigarrow S, h_1, \mathit{Norm}(\_) \models \mathbf{req}\ \sigma{\wedge}\pi & \mathit{iff}\ \ h_1{\subseteq}h \text{ and } S, h_1 \models \sigma{\wedge}\pi \\
S, h \rightsquigarrow S, h, \mathit{Err} \models \mathbf{req}\ \sigma{\wedge}\pi & \mathit{iff}\ \ \forall h_1 \cdot h_1{\subseteq}h \Rightarrow S, h_1 \not\models \sigma{\wedge}\pi \\
S, h \rightsquigarrow S, h, R \models \mathbf{req}\ (\sigma{\wedge}\pi)@R & \mathit{iff}\ \ S, h \rightsquigarrow S, h_1, R \models \mathbf{req}\ (\sigma{\wedge}\pi) \\
S, h \rightsquigarrow S, h{\circ}h_1{,}\mathit{Norm}(r) \models \mathbf{ens}[r]\ \sigma{\wedge}\pi & \mathit{iff}\ \ S, h_1 \models \sigma{\wedge}\pi \text{ and } \mathit{dom}(h_1){\cap}\mathit{dom}(h){=}\{\} \\
S, h \rightsquigarrow S_1, h_1, R \models f(x^*, r) & \mathit{iff}\ \ S(f) = \mathit{fun}\,(y^*)\,\Phi[r']{\rightarrow}\,e, \\
 & \quad S, h \rightsquigarrow S_1, h_1, R \models [r'{:=}r][y^*{:=}x^*]\Phi \\
S, h \rightsquigarrow S_1, h_1, R \models \exists\, x \cdot \Phi & \mathit{iff}\ \ \exists v \cdot S[x{:=}v], h \rightsquigarrow S_1, h_1, R \models \Phi \\
S, h \rightsquigarrow S_2, h_2, R \models \Phi_1; \Phi_2 & \mathit{iff}\ \ S, h \rightsquigarrow S_1, h_1, \mathit{Norm}(r) \models \Phi_1, \\
 & \quad S_1, h_1 \rightsquigarrow S_2, h_2, R \models \Phi_2 \\
S, h \rightsquigarrow S_1, h_1, \top \models \Phi_1; \Phi_2 & \mathit{iff}\ \ S, h \rightsquigarrow S_1, h_1, \top \models \Phi_1 \\
S, h \rightsquigarrow S_3, h_3, \mathit{Norm}(r_3) \models \Phi_1 \vee \Phi_2 & \mathit{iff}\ \ \exists h_1{,}h_2{,}r_1{,}r_2 \cdot S, h \rightsquigarrow S_1, h_1, \mathit{Norm}(r_1) \models \Phi_1 \\
 & \quad \text{and } S, h \rightsquigarrow S_2, h_2, \mathit{Norm}(r_2) \models \Phi_2, \text{ and} \\
 & \quad (S_3, h_3, r_3){\in}\{(S_1, h_1, r_1), (S_2, h_2, r_2)\} \\
S, h \rightsquigarrow S_1, h_1, \top \models \Phi_1 \vee \Phi_2 & \mathit{iff}\ \ S{,}h \rightsquigarrow S_1{,}h_1{,}\top \models \Phi_1\ \mathit{or}\ S{,}h \rightsquigarrow S_1{,}h_1{,}\top \models \Phi_2
\end{array}
$$

Fig. 5: Semantics of Staged Formulae

to an existential value v. Sequential composition $\Phi_1;\Phi_2$ results in a failure $\top$ if Φ_1 does, while disjunction $\Phi_1 \vee \Phi_2$ requires both branches not to fail.

3.2 Compaction

Staged formulae subsume separation logic triples, but triples suffice for many verification tasks, particularly those without calls to unknown functions, and we would like to recover their succinctness in cases where intermediate states are not required. This motivates a *compaction* or *normalization* procedure for staged formulae, written $\Phi \Longrightarrow \Phi$ (Fig. 6). Compaction is also useful for aligning staged formulae, allowing entailment proofs to be carried out stage by stage; we elaborate on this use in Section 5.

$$
\begin{array}{rcl}
\mathbf{ens}[\_]\ \mathit{false}\,; \Phi & \Longrightarrow & \mathbf{ens}[\_]\ \mathit{false} \\
\mathit{emp}; \Phi & \Longrightarrow & \Phi \\
\Phi; \mathit{emp} & \Longrightarrow & \Phi
\end{array}
\qquad
\begin{array}{c}
\mathbf{req}\ D_1\,; \mathbf{req}\ D_2 \Longrightarrow \mathbf{req}\ (D_1 * D_2) \\
\mathbf{ens}[\_]\ D_1\,; \mathbf{ens}[r]\ D_2 \Longrightarrow \mathbf{ens}[r]\ (D_1 * D_2) \\[1em]
\dfrac{D_A * D_1 \vdash D_2 * D_F}{\mathbf{ens}[r]\ D_1\,; \mathbf{req}\ D_2 \Longrightarrow \mathbf{req}\ D_A\,; \mathbf{ens}[r]\ D_F}
\end{array}
$$

Fig. 6: Select compaction rules

The three rules on the left simplify flows. A false postcondition ($\mathbf{ens}\ \sigma{\wedge}\mathit{false}$) models an unreachable or nonterminating program state, so the rest of a flow may be safely ignored. *emp* in the next two rules is *either* ($\mathbf{req}\ \mathit{emp}{\wedge}\mathit{true}$) or ($\mathbf{ens}\ \mathit{emp}{\wedge}\mathit{true}$); either may serve as an identity for flows. The first two rules on the right merge consecutive pre- and postconditions. Intuitively, they are sound because symbolic heaps separated by sequential composition must be disjoint to be meaningful – this follows from the use of disjoint union in Fig. 5. The last rule allows a precondition $\mathbf{req}\ D_2$ to be transposed with a preceding postcondition $\mathbf{ens}\ D_1$. This is done using biabduction [4], which computes a pair of antiframe D_A and frame D_F such that the antiframe is the new precondition required, and frame is what remains after proving the known precondition. The given rule assumes that D_1 and D_2 are disjoint[2]. A read-only $@R$ heap assertion under **req** would be handled by matching but not removing from D_F (see [8]).

Thus staged formulae can always be compacted into the following form, consisting of a disjunction of *flows* θ (a disjunction-free staged formula)[3], each consisting of a prefix of function stages (preceded by a description of the intermediate state at that point), followed by a final pre- and postcondition, capturing any behavior remaining after calling unknown functions.

$$\begin{aligned}&\Phi ::= \theta \mid \Phi \vee \Phi\\&\theta ::= (\exists\, x^* \cdot \mathbf{req}\ D; \exists\, x^* \cdot \mathbf{ens}[\_]\ D; f(v^*, r)\ ;)^*\ \exists\, x^* \cdot \mathbf{req}\ D; \exists\, x^* \cdot \mathbf{ens}[\_]\ D\end{aligned}$$

An example of compaction is given below (Fig. 7, left). We start at the first two stages of the flow and solve a biabduction problem (shown on the right, with solution immediately below) to infer a precondition for the whole flow, or, more operationally, to "push" the **req** to the left. We will later be able to rely on the new precondition to know that $a = 1$ when proving properties of the rest of the flow. Finally, we may combine the two **ens** stages because sequential composition guarantees disjointness. Normalization is *sound* in the sense that it transforms staged formulae without changing their meaning.

$$\begin{array}{lr}\quad\ \mathbf{ens}\ x \mapsto 1 * y \mapsto 2; \mathbf{req}\ x \mapsto a; \mathbf{ens}\ x \mapsto a{+}1 & \\ \Longleftarrow\!\!\!\Longrightarrow \mathbf{req}\ a{=}1; \mathbf{ens}\ y \mapsto 2; \mathbf{ens}\ x \mapsto a{+}1 & D_A * x \mapsto 1 * y \mapsto 2 \vdash x \mapsto a * D_F \\ \Longleftarrow\!\!\!\Longrightarrow \mathbf{req}\ a{=}1; \mathbf{ens}\ y \mapsto 2 * x \mapsto a{+}1 & D_A{=}(a{=}1),\ D_F{=}(y \mapsto 2)\end{array}$$

Fig. 7: An example of compaction

Theorem 1 (Soundness of Normalization). *Given* $\Phi_1 \Longleftarrow\!\!\!\Longrightarrow \Phi_2$*, if* $S, H \rightsquigarrow S_1, H_1, R_1 \models \Phi_1$*, then* $S, H \rightsquigarrow S_1, H_1, R_1 \models \Phi_2$*.*

Proof. By case analysis on the derivation of $\Phi_1 \Longleftarrow\!\!\!\Longrightarrow \Phi_2$. See the appendix [13].

4 Forward Rules for Staged Logics

To verify that a program satisfies a given specification Φ_s, we utilize a set of rules (presented in Fig. 8) to compute an abstraction or summary of the program Φ_p, then discharge the proof obligation $\Phi_p \sqsubseteq \Phi_s$ (covered in Section 5), in a manner similar to strongest postcondition calculations.

We make use of the following notations. _ denotes an anonymous existentially quantified variable. $[x{:=}v]\Phi$ denotes the substitution of x with v in Φ, giving

[2] More exhaustive aliasing scenarios are considered in the appendix [13].

[3] Using further normalization rules such as $(\Phi_1 \vee \Phi_2); \Phi_3 \Longleftarrow\!\!\!\Longrightarrow (\Phi_1; \Phi_3) \vee (\Phi_2; \Phi_3)$

$$\frac{\Phi_1 \sqsubseteq \Phi_3 \quad \{\ \Phi_3\ \}\ e\ \{\ \Phi_4\ \} \quad \Phi_4 \sqsubseteq \Phi_2}{\{\ \Phi_1\ \}\ e\ \{\ \Phi_2\ \}}\ \textbf{Consequence} \qquad \frac{\{\ \Phi_1\ \}\ e\ \{\ \Phi_2\ \}}{\{\ \Phi;\Phi_1\ \}\ e\ \{\ \Phi;\Phi_2\ \}}\ \textbf{Frame}$$

$$\frac{\mathit{fresh}\ \mathit{res}}{\{\ \Phi\ \}\ x\ \{\ \Phi\,;\mathbf{ens}[\mathit{res}]\ \mathit{res}{=}x\ \}}\ \textbf{Var} \qquad \frac{\mathit{fresh}\ \mathit{res}}{\{\ \Phi\ \}\ v\ \{\ \Phi\,;\mathbf{ens}[\mathit{res}]\ \mathit{res}{=}v\ \}}\ \textbf{Val (c | nil | } \mathbf{x_1{::}x_2)}$$

$$\frac{\mathit{fresh}\ r}{\{\ \Phi\ \}\ \mathit{ref}\ x\ \{\ \Phi\,;\mathbf{ens}[r]\ r \mapsto x\ \}}\ \textbf{Ref}$$

$$\frac{\mathit{fresh}\ a, \mathit{res}}{\{\ \Phi\ \}\ !x\ \{\ \Phi\,;\exists\, a \cdot \mathbf{req}\ x \mapsto a\,;\mathbf{ens}[\mathit{res}]\ x \mapsto a \wedge \mathit{res}{=}a\ \}}\ \textbf{Deref}$$

$$\frac{}{\{\ \Phi\ \}\ x_1{:=}x_2\ \{\ \Phi\,;\mathbf{req}\ x_1 \mapsto \_\,;\mathbf{ens}[\_]\ x_1 \mapsto x_2\ \}}\ \textbf{Assign}$$

$$\frac{\{\ \Phi;\mathbf{ens}[\_]\ x\ \}\ e_1\ \{\ \Phi_1[r_1]\ \} \quad \{\ \Phi;\mathbf{ens}[\_]\ \neg x\ \}\ e_2\ \{\ \Phi_2[r_2]\ \}}{\{\ \Phi\ \}\ \mathit{if}\ x\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \{\ \Phi_1 \vee [r_2{:=}r_1]\Phi_2\ \}}\ \textbf{If}$$

$$\frac{\mathit{fresh}\ x \quad \{\ \Phi\ \}\ e_1\ \{\ \Phi_1[r]\ \} \quad \{\ [r{:=}x]\Phi_1\ \}\ e_2\ \{\ \Phi_2\ \}}{\{\ \Phi\ \}\ \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2\ \{\ \exists\, x \cdot \Phi_2\ \}}\ \textbf{Let}$$

$$\frac{\mathit{fresh}\ \mathit{res} \quad \{\ \mathbf{ens}[\_]\ \mathit{Pure}(\Phi)\ \}\ e\ \{\ \Phi_p[r']\ \} \quad ([r'{:=}r]\Phi_p) \sqsubseteq \Phi_s}{\{\ \Phi\ \}\ \mathit{fun}\,(x^*)\ \Phi_s[r] \to e\ \{\ \Phi\,;\mathbf{ens}[\mathit{res}]\ \mathit{res}{=}\lambda\,(x^*, r) \to \Phi_s\ \}}\ \textbf{Lambda}$$

$$\frac{\mathit{fresh}\ \mathit{res}}{\{\ \Phi\ \}\ f(x^*)\ \{\ \Phi\,;f(x^*, \mathit{res})\ \}}\ \textbf{Call} \qquad \frac{}{\{\ \Phi\ \}\ \mathit{assert}\ D\ \{\ \Phi\,;\mathbf{req}\ D@R\ \}}\ \textbf{Assert}$$

Fig. 8: Forward Reasoning Hoare Rules with Staged Logics

priority to recently bound variables. We lift sequencing from flows to disjunctive staged formulae in the natural way: $\Phi_1\,;\Phi_2 \triangleq \bigvee\{\theta_1\,;\theta_2 \mid \theta_1 \in \Phi_2, \theta_2 \in \Phi_2\}$.

The first two rules in Fig. 8 are structural. The **Consequence** rule uses *specification subsumption* (detailed in Section 5) in place of implication – a form of behavioral subtyping. The **Frame** rule has both a *temporal* interpretation, which is that the reasoning rules are compositional with respect to the *history* of the current flow, and a *spatial* interpretation, consistent with the usual one from separation logic, if one uses the normalization rules (Section 3.2) to move untouched p from the final states of Φ_1 and Φ_2 into the frame Φ.

The **Val** rule illustrates how the results of pure expressions are tracked, in a distinguished *res* variable. The **Ref** rule results in a new, existentially-quantified location being added to the current state. The **Deref** and **Assign** rules are similar, both requiring proof that a named location exists with a value, then respectively either returning the value of the location and leaving it unchanged, or changing the location and returning the unit value. **Assert** checks its heap state without modifying it using the $@R$ read-only annotation. **If** is where disjunctive formula arises, and **Let** sequences expressions, renaming the intermediate result of e_1 accordingly. In the **Let** rule, program variable in staged formulae may extend scope of its corresponding variable in program code if no name clashes. It becomes existentially quantified when the local variable is out of scope.

The **Lambda** rule handles function definition annotated with a given specification Φ_s. The body of the lambda is summarized into Φ_p starting from pure information $Pure(\Phi)$ from its program context. Its behavior must be subsumed by the given specification. The result is then the lambda expression itself.

The **Call** rule is completely trivial, yet perhaps the most illuminating as to the design of *HSSL*. A standard modular verifier would utilize this rule to look up the specification associated with f, prove its precondition, then assume its postcondition. In our setting, however, there is the possibility that f is higher-order, unknown, and/or unspecified. Moreover, there is no need to prove the precondition of f immediately, due to the use of flows for describing program behaviors. Both of these point to the simple use of a function stage, which stands for a *possibly-unknown* function call. Utilizing the specification of f, if it is provided, is deferred to the unfolding done in the entailment procedure.

We prove soundness of these rules, which is to say that derived specifications faithfully overapproximate the programs they are derived from. In the following theorem, $e, h, S \rightsquigarrow h_1, S_1$ is a standard big-step reduction relation whose definition we leave to the appendix [13]. Termination is also considered in the appendix [13]. However, completeness is yet to be established.

Theorem 1 (Soundness of Forward Rules) *Given* $\{\ emp\ \}\ e\ \{\ \Phi\ \}$, *then* $\forall S, h, S_2, h_1 \cdot (S, h \rightsquigarrow S_2, h_1, Norm(r) \models \Phi) \Rightarrow \exists S_1 \cdot e, h, S \rightsquigarrow Norm(v), h_1, S_1$ *and* $S_1 \subseteq S_2$ *and* $S_1(r) = v$.

Proof. By induction on the derivation of $e, h, S_1 \rightsquigarrow R_1, h_1, S_1$. See appendix [13].

5 Staged Entailment Checking and its Soundness

In this section, we outline how entailments of the form $F \vdash \Phi_p \sqsubseteq \Phi_s$ may be automatically checked. F denotes heap and pure frames that are propagated by our staged logics entailment rules. Our entailment is always conducted over the compacted form where non-recursive staged predicate definitions are unfolded, while unknown predicates are matched exactly. Lemmas are also used to try re-summarize each instantiation of recursive staged predicates to simpler forms, where feasible. As staged entailment ensures that all execution traces that satisfy Φ_p must also satisfy Φ_s, we rely on theory of *behavioral subtyping* [19] to relate them. Specifically, we check that *contravariance holds* for pre-condition entailment, while *covariance holds* for post-condition entailment, as follows:

$$\frac{\mathit{fresh}\ y^* \qquad F_0 * D_2 \vdash (\exists\, x^*.\, D_1) * F \qquad F \vdash \theta_a \sqsubseteq \theta_c}{F_0 \vdash (\exists\, x^* \cdot \mathbf{req}\ D_1; \theta_a) \sqsubseteq (\exists\, y^* \cdot \mathbf{req}\ D_2; \theta_c)}\ \textbf{EntReq}$$

$$\frac{\mathit{fresh}\ x^* \qquad F_0 * D_1 \vdash (\exists\, y^*.\, D_2) * F \qquad F \vdash \theta_a \sqsubseteq \theta_c}{F_0 \vdash (\exists\, x^* \cdot \mathbf{ens}[r]\ D_1; \theta_a) \sqsubseteq (\exists\, y^* \cdot \mathbf{ens}[r]\ D_2; \theta_c)}\ \textbf{EntEns}$$

More details of staged entailment rules are given in the appendix [13]. Note that we use another entailment over separation logic $D_1 \vdash D_2 * F_r$ that can propagate residual frame, F_r. Lastly, we outline the soundness of staged entailemt against the semantics of staged formulae, ensuring that all derivations are valid.

Theorem 2 (Soundness of Entailment). *Given* $\Phi_1 \sqsubseteq \Phi_2$ *and* $S, h \rightsquigarrow Norm(r_1), S_1, h_1 \models \Phi_1$*, then there exists* h_2 *such that* $S, h \rightsquigarrow Norm(r_1), S_2, h_2 \models \Phi_2$ *where* $h_2 \subseteq h_1$. (Here, $h_1 \subseteq h_2$ denotes that $\exists\, h_3.\, h_1 \circ h_3 = h_2$.)

Proof. By induction on the derivation of $\Phi_1 \sqsubseteq \Phi_2$. See the appendix [13].

6 Implementation and Initial Results

We prototyped our verification methodology and techniques in a tool named Heifer. Our tool takes input programs written in a subset of OCaml annotated with user-provided specifications. It analyzes input programs to produce normalized staged formulae (Section 3.2, Section 4), which it then translates to first-order verification conditions (Section 5) suitable for an off-the-shelf SMT solver. Here, our prototype targets SMT encodings via Why3. As an optimization, it uses Z3 directly for queries which do not require Why3's added features.

Table 1: A Comparison with Cameleer and Prusti. (Programs that are natively inexpressible are marked with "✗". Programs that cannot be reproduced from Prusti's artifact [1] are marked with "-" denoting incomparable. We use T to denote the total verification time (in seconds) and T_P to record the time spent on the external provers.)

	Heifer				Cameleer [23]			Prusti [32]		
Benchmark	LoC	LoS	T	T_P	LoC	LoS	T	LoC	LoS	T
map	13	11	0.66	0.58	10	45	1.25		-	
map_closure	18	7	1.06	0.77		✗			-	
fold	23	12	1.06	0.87	21	48	8.08		-	
fold_closure	23	12	1.25	0.89		✗			-	
iter	11	4	0.40	0.32		✗			-	
compose	3	1	0.11	0.09	2	6	0.05		-	
compose_closure	23	4	0.44	0.32		✗			✗	
closure [28]	27	5	0.37	0.27		✗		13	11	6.75
closure_list	7	1	0.15	0.09		✗			-	
applyN	6	1	0.19	0.17	12	13	0.37		-	
blameassgn [12]	14	6	0.31	0.28		✗		13	9	6.24
counter [18]	16	4	0.24	0.18		✗		11	7	6.37
lambda	13	5	0.25	0.22		✗			-	
	197	73			45	112		37	27	

We have verified a suite of programs (Table 1) involving higher-order functions and closures. As the focus of our work is to explore a new program logic and subsumption-based verification methodology (rather than to verify existing programs), the benchmarks are small in size, and are meant to illustrate the style of specification and give a flavor of the potential for automation.

Table 1 provides an overview of the benchmark suite. The first two subcolumns show the size of each program (LoC) and the number of lines of user-provided specifications (LoS) required. The next two give the total wall-clock time taken (in seconds) to verify all functions in each program against the provided specifications, and the amount of time spent in external provers.

The next column shows the same programs verified using Cameleer [23, 26], a state-of-the-art deductive verifier. Cameleer serves as a good baseline for several reasons: it is representative of the dominant paradigm of pre/post specifications and, like Heifer, targets (a subset of) OCaml. It supports higher-order functions in both programs and specifications [27]. The most significant differences between Cameleer and Heifer are that Cameleer does not support *effectful* higher-order functions and is intended to be used via the Why3 IDE in a semi-interactive way (allowing tactic-like *proof transformations*, used in the above programs).

The last column shows results for Prusti [32]. Despite Rust's ownership type system, we compare it against Prusti because of its state-of-the-art support for mutable closures, highlighting differences below. While we were able to reproduce the claims made in Prusti's OOPSLA 2021 artifact [1], we were not able to verify many of our own benchmark programs due to two technical reasons, namely lacking support for Rust's `impl Trait` (to return closures) and ML-like cons lists (which caused timeouts and crashes). Support for closures is also not yet in mainline Prusti [2]. Nevertheless, we verified the programs we could use for the artifact, the results of which are shown in Table 1. All experiments were performed on macOS using a 2.3 GHz Quad-Core Intel Core i7 CPU with 16 GB of RAM. Why3 1.7.0 was used, with SMT solvers Z3 4.12.2, CVC4 1.8, and Alt-Ergo 2.5.2. The Prusti artifact, a Docker image, was run using Moby 25.0.1.

User annotations required. Significantly less specification than code is required in Heifer, with an average LoS/LoC ratio of 0.37. This is helped by two things: the use of function stages in specifications, and the use of biabduction-based normalization, which allows the specifications of functions to be mostly automated, requiring only properties and auxiliary lemmas to be provided. In contrast, Cameleer's ratio is 2.49, due to the need to adequately summarize the behaviors of the function arguments and accompany these summaries with invariants and auxiliary lemmas. Two examples illustrating this are detailed in the appendix [13]. Prusti's ratio is 0.73, but a caveat is that in the programs for it, only closure reasoning was used, without lemmas or summarization.

Expressiveness. Heifer is able to express many programs that Cameleer cannot, particularly closure-manipulating ones. This accounts for the ✗ rows in Table 1. While some of these can be verified with Prusti, unlike stages, Prusti's call descriptions do not capture ordering [10, 1]; an explicit limitation as shown by the ✗ rows in Prusti's column. Prusti is able to use history invariants and the ownership of the Rust type system, but this difference is more than mitigated in Heifer with the adoption of an expressive staged logic with spatial heap state; more appropriate for the weaker (but more general) type system of OCaml.

7 Related Work

The use of sequential composition in specifications goes back to classic theories of program refinement, such as Morgan's refinement calculus [20] and Hoare and He's Unifying Theories [15], as well session types [9] and logics [7]. It has also been used to structure verification conditions and give users control over the order in which they are given to provers [14], allowing more reliable proof automation. We extend both lines of work, developing the use of sequential composition as a precise specification mechanism for higher-order imperative functions, and using it to guide entailment proofs of staged formulae.

Higher-order imperative functions were classically specified in program logics using *evaluation formulae* [16] and *reference-reachability predicates* [34]. The advent of separation logic has allowed for simpler specifications using *invariants* and *nested triples* (Section 1). These techniques are common in higher-order separation logics, such as HTT [21], CFML [5], Iris [17] and Steel/Pulse [30], which are encoded in proof assistants (e.g. Coq, F$\star$ [29]) which do not natively support closures or heap reasoning. While the resulting object logics are highly expressive, they are much more complex (owing to highly nontrivial encodings) and consequently less automated than systems that discharge obligations via SMT. We push the boundaries in this area by proposing stages as a new, precise specification mechanism which is compatible with automated verification.

The guarantees of an expressive type system can significantly simplify how higher-order state is specified and managed. Prusti [32] exploits this with *call descriptions* (an alternative to function stages, as pure assertions saying that a call has taken place with a given pre/post) and *history invariants*, which rely on the ownership of mutable locations that closures have in Rust. Creusot [10] uses a *prophetic mutable value semantics* to achieve a similar goal with pre/post specifications of closures. Our solution is not dependent on an ownership type system, applying more generally to languages with unrestricted mutation.

Defunctionalization [24] is another promising means of reasoning about higher-order effectful programs [27], pioneered by the Why3 [11]-based Cameleer [23]. This approach currently inherits the latter's lack of native support for closures.

Our approach to automated verification is currently based on strict evaluation. It would be interesting to see how staged specification can be extended to support verification of lazy evaluation, as had been explored in [33] and [31].

8 Conclusion

We have explored how best to *modularly specify and verify higher-order imperative programs*. Our contributions are manifold: we propose a new staged specification logic, rules for deriving staged formulae from programs and normalizing them using biabduction, and an entailment proof system. This forms the basis of a new verification methodology, which we validate with our prototype Heifer.

To the best of the authors' knowledge, this work is the first to introduce a *fundamental* staged specification mechanism for verifying higher-order imperative programs *without* any presumptions; being *more concise* (without the need for specifying abstract properties) and *more precise* (without imposing preconditions on function-typed parameters) when compared to existing solutions.

Acknowledgments This research is supported by the Ministry of Education, Singapore, under the Academic Research Fund Tier 1 (FY2023) (Project Title: Automated Verification for Imperative Higher-Order Programs).

Bibliography

[1] Modular specification and verification of closures in Rust (artefact), 2021.

[2] (documentation of) closures. `https://github.com/viperproject/prusti-dev/issues/1431`, 2024.

[3] Fernando Alves, Delano Oliveira, Fernanda Madeiral, and Fernando Castor Filho. On the bug-proneness of structures inspired by functional programming in JavaScript projects. *ArXiv*, abs/2206.08849, 2022.

[4] Cristiano Calcagno, Dino Distefano, Peter W O'hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM (JACM)*, 58(6):1–66, 2011.

[5] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011.

[6] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. *ACM SIGPLAN Notices*, 43(1):87–99, 2008.

[7] Andreea Costea, Wei-Ngan Chin, Shengchao Qin, and Florin Craciun. Automated modular verification for relaxed communication protocols. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 284–305. Springer, 2018.

[8] Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2011.

[9] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012.

[10] Xavier Denis and Jacques-Henri Jourdan. Specifying and verifying higher-order Rust iterators. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2023.

[11] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *European Symposium on Programming*, 2013.

[12] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[13] Darius Foo, Yahui Song, and Wei-Ngan Chin. Staged Specifications for Automated Verification of Higher-Order Imperative Programs. Technical report, National University of Singapore. DOI:arxiv-2308.00988, 2023.

[14] Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 386–401. Springer, 2011.

[15] Jifeng He and Charles Antony Richard Hoare. Unifying theories of programming. In *RelMiCS*, 1998.

[16] Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order frame rules. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 270–279. IEEE Computer Society, 2005.

[17] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[18] Ioannis T. Kassios and Peter Müller. Specification and verification of closures. 2010. unpublished.

[19] Gary T Leavens and David A Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):1–88, 2015.

[20] Carroll Morgan. The refinement calculus. In *NATO ASI PDC*, 1994.

[21] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.

[22] Peter W O'Hearn. A primer on separation logic (and automatic program verification and analysis). *Software safety and security*, 33:286–318, 2012.

[23] Mário Pereira and António Ravara. Cameleer: A deductive verification tool for ocaml. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021.

[24] John C Reynolds. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740, 1972.

[25] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.

[26] Tiago Soares and Mário Pereira. A framework for the automated verification of algebraic effects and handlers (extended version). *ArXiv*, abs/2302.01265, 2023.

[27] Tiago Lopes Soares. A Deductive Verification Framework for Higher Order Programs. *arXiv preprint arXiv:2011.14044*, 2020.

[28] Kasper Svendsen. *Modular specification and verification for higher-order languages with state*. IT-Universitetet i København, 2013.

[29] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[30] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. *Proceedings of the ACM on Programming Languages*, 4:1 – 30, 2020.

[31] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: experience with refinement types in the real world. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51. ACM, 2014.

[32] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021.

[33] Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 41–52. ACM, 2009.

[34] Nobuko Yoshida, Kohei Honda, and Martin Berger. Logical reasoning for higher-order functions with local state. In Helmut Seidl, editor, *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007, Proceedings*, volume 4423 of *Lecture Notes in Computer Science*, pages 361–377. Springer, 2007.

[35] Fiorella Zampetti, Francois Belias, Cyrine Zid, and Giuliano Antonioland Massimiliano Di Penta. An empirical study on the fault-inducing effect of functional constructs in Python. *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 47–58, 2022.

A Inferred specifications

Inferred specifications for the examples in the paper.

Pure predicates that can also be proven terminating are shown below. One of them, namely *range*, has a precondition to ensure termination.

$$\begin{aligned}
\mathit{length}(\mathit{xs}, \mathit{res}) = {} & \mathit{xs}{=}[]\wedge \mathit{res}{=}0 \\
& \vee \exists\, r, x, \mathit{xs}_1 \cdot \mathit{xs}{=}x{::}\mathit{xs}_1 \wedge \mathit{length}(\mathit{xs}_1, r) \wedge \mathit{res}{=}r{+}1 \\
\mathit{incrg}(\mathit{init}, \mathit{li}, \mathit{res}) = {} & \mathit{li}{=}[]\wedge \mathit{res}{=}[] \\
& \vee \exists\, r, x, \mathit{xs}_1 \cdot \mathit{li}{=}x{::}\mathit{xs}_1 \wedge \mathit{incrg}(\mathit{init}{+}1, \mathit{xs}_1, r) \wedge \mathit{res}{=}\mathit{init}{::}r \\
\mathit{inc}(x, \mathit{res}) = {} & \mathit{res}{=}x{+}1 \\
\mathit{range}(x, n, \mathit{res}) = {} & \mathbf{req}\ n{\geq}0; \\
& \mathbf{ens}[\_]\ n{=}0 \wedge \mathit{res}{=}[]\ \ \vee\ \ \exists\, r \cdot n{>}0 \wedge \mathit{range}(x{+}1, n{-}1, r) \wedge \mathit{res}{=}x{::}r
\end{aligned}$$

Note that the precondition to ensure termination of each pure predicate would have been baked into the pure predicate's definition itself. For example, $\mathit{range}(x, n, \mathit{res}) \Rightarrow n{\geq}0$.

Below are staged specifications for higher-order methods that have been inferred. Since the unknown function parameter may be imperative, these higher-order functions are currently only expressible in the staged specification format.

$$\begin{aligned}
\mathit{map}(f,\ \mathit{xs},\ \mathit{res}) = {} & \mathbf{ens}[\_]\ \mathit{xs}{=}[] \wedge \mathit{res}{=}[] \\
& \vee \exists\, h, x, \mathit{xs}_1 \cdot \mathbf{ens}[\_]\ \mathit{xs}{=}x{::}\mathit{xs}_1; f(x, h); \exists\, t \cdot \mathit{map}(f, \mathit{xs}_1, t); \\
& \quad \mathbf{ens}[\_]\ \mathit{res}{=}h{::}t \\
\mathit{applyN}(f, x, n, \mathit{res}) = {} & \mathbf{ens}[\_]\ n{=}0 \wedge \mathit{res}{=}x \\
& \vee \mathbf{ens}[\_]\ n{\neq}0; \exists\, r \cdot f(x, r); \mathit{applyN}(f, r, n{-}1, \mathit{res}) \\
\mathit{take}(f, n, \mathit{res}) = {} & \mathbf{ens}[\_]\ n{=}0 \wedge \mathit{res}{=}[] \\
& \vee \mathbf{ens}[\_]\ n{\neq}0; \exists\, r, r2 \cdot f(r); \mathit{take}(f, n{-}1, r2); \mathbf{ens}[\_]\ \mathit{res}{=}r{::}r2
\end{aligned}$$

Below is a staged specification for a first-order imperative method that has been inferred. Two-staged pre/post specifications are always possible for first-order imperative method that returns normally.

$$\mathit{integers}(\mathit{res}) = \ \exists\, a \cdot \mathbf{req}\ x \mapsto a; \mathbf{ens}[\_]\ x \mapsto \mathit{res} \wedge \mathit{res}{=}a{+}1$$

B Details on Mutable Closures

We provide some verification details on an effectful higher-order function in Fig. 9. Here *counter* is a function that captures a heap-allocated location x, which is no longer in scope by the point *counter* is invoked. The specification $\lambda(\mathit{res})$ (line 4) compositionally describes the behavior of the lambda expression in *counter*: it updates the value of a known location x which initially has some value i, which it returns. Reasoning about its enclosing let-expression (line 3),

```
1  let mut_closure () $^\circ m\_c(res) = \exists x \cdot \textbf{ens}[\_]\ x \mapsto 2 \wedge res{=}1$ =
2    let counter =
3      let x = ref 0 in
4      fun () $^\circ\lambda(res)\ \exists i \cdot \textbf{req}\ x \mapsto i; \textbf{ens}[\_]\ x \mapsto i{+}1 \wedge res{=}i$ ->
5        let r = !x in x := !x + 1; r
6    in
7    $^\circ\{\ \exists x \cdot \textbf{ens}[\_]\ x \mapsto 0 \wedge counter(res){=}\exists i \cdot \textbf{req}\ x \mapsto i; \textbf{ens}[\_]\ x \mapsto i{+}1 \wedge res{=}i\ \}$
8    counter ();
9    $^\circ\{\ \exists x \cdot \textbf{ens}[\_]\ x \mapsto 0 \wedge counter(res){=}\cdots; \exists r \cdot counter(r)\ \}$
10   $^\circ\{\ \exists x, r \cdot \textbf{ens}[\_]\ x \mapsto 1 \wedge counter(res){=}\cdots \wedge r{=}0\ \}$
11   counter ()
12   $^\circ\{\ \exists x, r, counter \cdot \textbf{ens}[\_]\ x \mapsto 1 \wedge counter(res){=}\cdots \wedge r{=}0; counter(res)\ \}$
13   $^\circ\{\ \exists x, r, counter \cdot \textbf{ens}[\_]\ x \mapsto 2 \wedge counter(res){=}\cdots \wedge r{=}0 \wedge res{=}1\ \}$
14   $^\circ\{\ \exists x \cdot \textbf{ens}[\_]\ x \mapsto 2 \wedge res{=}1\ \}$
```

Fig. 9: A higher-order function using a mutable closure

however, we see that x is a new location, distinct from any other, resulting in it being existentially quantified. This quantifier ensures that x may only be modified by calling *counter*, and surfaces in the specification m_c at line 1, allowing us to precisely describe its result.

For modularity, we might wish to hide the persistence of locations created by *mut_closure* (e.g., assuming the presence of a garbage collector). We can do this simply by using the specification $\textbf{ens}[\_]\ res{=}1$, which is an (intuitionistic) weakening of m_c.

A subtlety about $\lambda(res)$ is that it does not assume anything about the value of x, requiring only that x exists as a location. Thus, even if we swapped lines 2 and 3, changing the scope of x and allowing the rest of *mut_closure*to mutate it, $\lambda(res)$ would be unchanged. This is in contrast to other systems which require additional guarantees to specify the lambda modularly; an example is Prusti [32], which leverages the ownership guarantees provided by the Rust type system.

C Additional Examples

This section may be read as an extension to Section 2. Here we highlight a few more interesting and involved example programs.

Closure-local state Fig. 10a shows a simple extension of Fig. 9. The internal states of multiple closures may be faithfully modelled using staged formulae. Again, the existential quantifiers serve both to restrict access to both x locations (renamed to i and j) and as a means of enforcing modularity in specifications. Specification subsumption provides a way to hide them, if the internal state of the closure does not matter in a particular context.

```
let two_closures ()
$^\circ\exists i, j \cdot \mathbf{ens}[\_]\ i \mapsto 1 * j \mapsto 2 \wedge res{=}3$
= let f = let x = ref 0 in
    fun () -> x:=!x+1; !x
   $^\circ\{\exists x \cdot \mathbf{ens}[\_]\ x \mapsto 0 \wedge f(r){=}\exists a \cdot$
        $\mathbf{req}\ x \mapsto a; \mathbf{ens}[\_]\ x \mapsto r \wedge r{=}a{+}1\}$
  in let g = let x = ref 0 in
         fun () -> x:=!x+2; !x
   $^\circ\{\exists x, x_2 \cdot \mathbf{ens}[\_]\ x \mapsto 0 * x_2 \mapsto 0 \wedge f(r){=}\cdots\wedge$
        $g(r){=}\exists a_2 \cdot \mathbf{req}\ x_2 \mapsto a_2;$
              $\mathbf{ens}[\_]\ x_2 \mapsto r \wedge r{=}a_2{+}2\}$
  in f()+g()
   $^\circ\{\exists x, x_2 \cdot \mathbf{ens}[\_]\ x \mapsto 0 * x_2 \mapsto 0 \wedge f(r){=}\cdots\wedge$
        $g(r){=}\cdots; \exists r_1, r_2 \cdot f(r_1); g(r_2);$
                   $\mathbf{ens}[\_]\ res{=}r_1{+}r_2\}$
   $^\circ\{\exists x, x_2, r_1, r_2 \cdot \mathbf{ens}[\_]\ x \mapsto 1 * x_2 \mapsto 0 \wedge f(r){=}\cdots$
        $\wedge g(r){=}\cdots \wedge r_1{=}1; g(r_2); \mathbf{ens}[\_]\ res{=}r_1{+}r_2\}$
   $^\circ\{\exists x, x_2, r_1, r_2 \cdot \mathbf{ens}[\_]\ x \mapsto 1 * x_2 \mapsto 2 \wedge f(r){=}\cdots$
        $\wedge g(r){=}\cdots \wedge r_1{=}1 \wedge r_2{=}2 \wedge res{=}r_1{+}r_2\}$
```

(a) Multiple closures

```
let rec range x n =
$^\circ range(x, n, res) = \mathbf{req}\ n{\geq}0; \cdots$
  if n = 0 then []
  else x::range (x+1) (n-1)

let rec take f n =
$^\circ take(f, n, res) = \cdots$
  if n = 0 then []
  else f()::take f (n-1)

let gen_contents x n
$gen\_contents(x, n, res) = \exists i \cdot \mathbf{req}\ x \mapsto i$
    $; \mathbf{ens}[\_]\ x \mapsto i{+}n \wedge range(i, n, res)$
= let integers =
   fun () -> let r = !x in
             x := !x + 1; r
  in take integers n
```

(b) Closures as generators

Fig. 10: More uses of stateful closures

Closures as generators With mutable closures, we can build *generators* – "next-element" functions of type `unit` → `'a` which represent potentially infinite sequences. In Fig. 10b, we recast *counter* (from Fig. 9) as the generator *integers*. We then prove a lemma *gen_contents* that uses the ghost functions *range* and *take* that says what *integers* contains, *for all n*. With only the given specification, this proof goes through automatically.

D Intersection of Staged Specifications

Intersection specifications are meant to capture different call scenarios and have an opposite semantics from disjunctions from conditional expressions. They currently arise primarily from a more exhaustive consideration of aliasing scenarios and it can be used to support proof search over disjoint preconditions.

An example of how intersection specifications may originate is from the following more complete consideration from bi-abduction rule:

$$\frac{\bigwedge\ (D_A * D_1 \vdash D_2 * D_F)}{\mathbf{ens}[\_]\ D_1 \,; \mathbf{req}\ D_2 \iff \bigwedge_{sp}\ (\mathbf{req}\ D_A \,; \mathbf{ens}[\_]\ D_F)}\ \textbf{Float Pre}$$

A concrete application is the following where x and y may be aliases or not.

$$\mathbf{ens}[\_]\ x \mapsto a \,; \mathbf{req}\ y \mapsto b \iff (\mathbf{req}\ y \mapsto b; \mathbf{ens}[\_]\ x \mapsto a) \wedge_{sp} (\mathbf{req}\ x{=}y \wedge a{=}b)$$

We use the symbol $\wedge_{sp}$ to distinguish it from $\vee$ that results from conditional constructs. Our normalization rules would place $\wedge_{sp}$ at outermost construct so that they may be used to support proof search during forward verification.

$$\Phi ::= \Phi' \mid \Phi \wedge_{sp} \Phi$$
$$\Phi' ::= \theta \mid \Phi' \vee \Phi'$$

An example of its use is the following more general staged specification of *hello* example from Sec 2.1 where $\wedge_{sp}$ is used in Stage 4a/4b.

$$\begin{array}{ll}
\mathit{hello}(f,x,y,\mathit{res}) = & \\
\quad \exists\, a \cdot \mathbf{req}\ x \mapsto a; & //\ \text{Stage 1: requiring } x \text{ be pre-allocated} \\
\quad \mathbf{ens}[\_]\ x \mapsto a{+}1; & //\ \text{Stage 2: ensuring } x \text{ is updated} \\
\quad \exists\, r \cdot f(y,r); & //\ \text{Stage 3: unknown higher-order } f \text{ call} \\
\quad \exists\, b \cdot\ (\ \mathbf{req}\ x \mapsto b * y \mapsto \_; & //\ \text{Stage 4a: requiring } x,\ y \text{ be pre-allocated} \\
\quad \mathbf{ens}[\_]\ x \mapsto b * y \mapsto \mathit{res} \wedge \mathit{res}{=}b{+}r & //\ \text{Stage 5a: } y \text{ is updated, and } x \text{ is unchanged} \\
\quad \wedge_{sp}\ \mathbf{req}\ x \mapsto b \wedge x{=}y; & //\ \text{Stage 4b: requiring } x \text{ be pre-allocated with } x = y \\
\quad\quad \mathbf{ens}[\_]\ x \mapsto \mathit{res} \wedge \mathit{res}{=}b{+}r\ ) & //\ \text{Stage 5b: } y \text{ (and hence } x\text{) is updated}
\end{array}$$

The intersection operator $\wedge_{sp}$ can be floated outermost so that its new staged specifications below can support proof search via Hoare-style verification rules.

$$\begin{array}{l}
\mathit{hello}(f,x,y,\mathit{res}) = \\
\quad \exists\, a \cdot \mathbf{req}\ x \mapsto a \wedge x{\neq}y; \mathbf{ens}[\_]\ x \mapsto a{+}1;\ \exists\, r \cdot f(y,r); \\
\quad \exists\, b \cdot\ \mathbf{req}\ x \mapsto b * y \mapsto \_;\ \mathbf{ens}[\_]\ x \mapsto b * y \mapsto \mathit{res} \wedge \mathit{res}{=}b{+}r \\
\quad \wedge_{sp}\ \exists\, a \cdot \mathbf{req}\ x \mapsto a \wedge x{=}y; \mathbf{ens}[\_]\ x \mapsto a{+}1;\ \exists\, r \cdot f(y,r); \\
\quad\quad \exists\, b \cdot\ \mathbf{req}\ x \mapsto b; \mathbf{ens}[\_]\ x \mapsto \mathit{res} \wedge \mathit{res}{=}b{+}r
\end{array}$$

E An Example to Illustrate the Forward Rules

As an example to illustrate the forward (Hoare) rules, consider the program in Fig. 11, annotated with intermediate specifications generated by the rules. It defines a location x that is captured by a closure and mutates x through the closure, asserting something about the closure's result at the end.

Line 2 is the result of applying the **Ref** rule, resulting in a new, existentially-quantified location x. Next, we reason about the lambda expression, starting with an empty history. Inferring a specification for the assignment involves first applying with **Deref** rule, then **Val** (taking + as primitive), then **Assign**, resulting in line 5. This is then normalized into the specification at line 6. The **Deref** rule is applied again, and the result is normalized.

Saving the specification of the lambda expression Φ_f for later, we continue outside it to line 11, using the **Call** rule twice, followed by the **Assert** rule. The result at line 15 is the inferred specification of the entire program and cannot be normalized further.

With this done, the next step might be to prove that this specification is subsumed by another specification that a user might write, e.g., $\exists\, x \cdot \mathbf{ens}[\_]\ x \mapsto 2 \wedge \mathit{res}{=}2$. In the next section, we detail a procedure to do this.

```
let x = ref 0 in
°{ $\mathbf{ens}[\_]\ x \mapsto 0$ }
let f = fun () ->
  x := !x + 1;
  °{ $\exists a \cdot \mathbf{req}\ x \mapsto a; \mathbf{ens}[\_]\ x \mapsto a \wedge res{=}a; \mathbf{ens}[\_]\ res =$
      $a{+}1; \mathbf{req}\ x \mapsto \_; \mathbf{ens}[\_]\ x \mapsto a{+}1 \wedge res{=}()$ }
  °{ $\exists a \cdot \mathbf{req}\ x \mapsto a; \mathbf{ens}[\_]\ x \mapsto a{+}1 \wedge res{=}()$ }
  !x
  °{ $\exists a \cdot \mathbf{req}\ x \mapsto a; \mathbf{ens}[\_]\ x \mapsto a{+}1; \exists b \cdot \mathbf{req}\ x \mapsto b; \mathbf{ens}[\_]\ x \mapsto b \wedge res{=}b$ }
  °{ $\exists a \cdot \mathbf{req}\ x \mapsto a; \mathbf{ens}[\_]\ x \mapsto res \wedge res{=}a{+}1$ }
in
°{ $\mathbf{ens}[\_]\ x \mapsto 0 \wedge f(res){=}\exists a \cdot \mathbf{req}\ x \mapsto a; \mathbf{ens}[\_]\ x \mapsto res \wedge res{=}a{+}1$ }
f ();
°{ $\mathbf{ens}[\_]\ x \mapsto 0 \wedge f(res){=}\cdots; \exists r_1 \cdot f(r_1)$ }
°{ $\mathbf{ens}[\_]\ x \mapsto 1 \wedge f(res){=}\cdots$ }
let r = f() in
°{ $\mathbf{ens}[\_]\ x \mapsto 1 \wedge f(res){=}\cdots; f(r)$ }
°{ $\mathbf{ens}[\_]\ x \mapsto 2 \wedge f(res){=}\cdots \wedge r{=}2$ }
assert (r = 2);
°{ $\mathbf{ens}[\_]\ x \mapsto 1 \wedge f(res){=}\cdots \wedge r{=}2; \mathbf{req}\ r{=}2$ }
°{ $\mathbf{ens}[\_]\ x \mapsto 1 \wedge f(res){=}\cdots \wedge r{=}2$ }
r
°{ $\mathbf{ens}[\_]\ x \mapsto 2 \wedge f(res){=}\cdots \wedge r{=}2; \mathbf{ens}[\_]\ res{=}r$ }
°{ $\mathbf{ens}[\_]\ x \mapsto 2 \wedge f(res){=}\cdots \wedge r{=}2 \wedge res{=}r$ }
°{ $\exists x, f, r \cdot \mathbf{ens}[\_]\ x \mapsto 2 \wedge f(res){=}\cdots \wedge r{=}2 \wedge res{=}r$ }
°{ $\exists x \cdot \mathbf{ens}[\_]\ x \mapsto 2 \wedge res{=}2$ }
```

Fig. 11: Forward rules example

F Comparing with User Annotations in Cameleer

Of the programs that can be handled, Heifer requires much less specification. As an example, Fig. 12 illustrate Cameleer specifications for *map* and *foldr*, which may be compared to the programs given in Section 2.

Because of the need to summarize the effects of f, *map*'s postcondition uses a quantifier (over sequence indices) to talk about the elements of the input and output lists. This then necessitates lemmas such as *index_shift* for relating indices to list destructuring, finally requiring more lines of specification per line of code. The *foldr* specification is similar to the one for Iris (Section 1), but does not use a higher-order triple, instead requiring a ghost argument for an invariant that must be preserved between calls to f. This approach is representative of many verifiers, including Dafny, WhyML, and vanilla F$\star$. As mentioned before, this parameterization of the specification with a summary of f is nontrivial, in that it cannot be mechanically done for every higher-order function, as this pair of examples shows.

```
let rec map (f:'a -> 'b) (xs:'a list) =
  match xs with
  | [] -> []
  | x :: xs1 -> f x :: map f xs1
(*@ ys = map f xs
      variant xs
      ensures length ys = length xs
      ensures forall i. 0 <= i < length ys ->
                   ys[i] = f (xs[i]) *)

(*@ lemma index_shift: forall x:'a, xs:'a list, i:int.
   1 <= i /\ i < length (Cons x xs) ->
     (Cons x xs)[i] = xs[i-1] *)

let rec foldr ((inv : 'b -> 'a seq -> bool) [@ghost])
  (f : 'a -> 'b -> 'b) (xs : 'a list) (acc : 'b)
= match xs with
  | [] -> acc
  | x :: t -> f x (foldr inv f t acc)
(*@ r = foldr inv f xs acc
      requires inv acc []
      requires forall acc x ys.
                 inv acc ys -> inv (f x acc) (cons x ys)
      variant  xs
      ensures  inv r xs *)
```

Fig. 12: *foldr* and *map* in Cameleer [23]

G Entailment for Staged Specifications

Our entailment over staged specification is always conducted over the compacted form where non-recursive staged predicate definitions are unfolded and compacted, while unknown predicates are always matched exactly. Lemmas are also also used to try re-summarize each instantiation of recursive staged predicates.

G.1 Staged (Specification) Subsumption

We assume staged formulae Φ are normalized, i.e., in the form described at the end of Section 3.2. Normalization both *aligns* and *compacts* staged formulae and is a crucial ingredient to the entailment procedure, which interleaves normalization and rewriting to prove subsumption stage by stage. We illustrate this proof system formally in Fig. 13, and by example at the end of this section.

The rules **DisjLeft** and **DisjRight** reduce disjunctive subsumption to subsumption between flows $F \vdash \theta_i \sqsubseteq \theta_j$, where F is an assumption $\sigma \wedge \pi$ which arises from the propagation of separation logic frames; where unspecified, it is $emp \wedge true$.

The next three rules check subsumption between the individual stages of flows, matching them pairwise before continuing on their tails. This matching is possible because flows are normalized. The rules **EntReq** and **EntEns** respectively express the contravariance of **req** preconditions and covariance of **ens** postconditions. Subsumption between stages reduces to standard separation logic entailments of the form $D_1 \vdash D_2 * F$, where F is an inferred frame, and all heaps that satisfy D_1 also satisfy D_2. Notably, inferred frames F propagate forward and become assumptions in the separation logic entailments of next stage; a motivating example is the subsumption $\mathbf{req}\ x \mapsto 1; \mathbf{ens}[\_]\ x \mapsto 2 \sqsubseteq \mathbf{req}\ x \mapsto 1 * F; \mathbf{ens}[\_]\ x \mapsto 2 * F$ for any F. The propagation of frames from **req** to **ens** stages has previously been called *enhanced specification subsumption* [6, Section 2.4], and has been historically used for verification of object-oriented programs; we extend this use to **ens** and **req** in multi-stage specifications.

The rule **EntFunc** requires that function constructors f match, and their arguments and return value are provably equal under the pure assumptions in F_0. Something notable about **EntFunc** is that only the pure portion of the frame propagates further across function stages, as the effects of instantiated function stages may invalidate any assumptions about the heap. Heap frames may thus be dropped if we are using intuitionistic separation logic. If classical separation logic is adopted, our subsumption procedure will need to enforce $\sigma{=}emp$ at the start of **EntRule** and at the end of our specification subsumption procedure.

Separation logic entailments are then reduced into first-order implication using the so-called "crunch, crunch" approach [22], via the next three rules: **SLMatch** reduces matching locations on both sides into equalities on their contents, **SLMatchEx** does the same for existentially-quantified locations (matching locations regardless of name to instantiate existentials, and possibly requiring backtracking), and **SLBase** serves as the base case, at which point the frame is

$$\frac{\theta_1 \sqsubseteq \Phi_c \qquad \theta_2 \sqsubseteq \Phi_c}{(\theta_1 \vee \theta_2) \sqsubseteq \Phi_c}\ \textbf{DisjLeft} \qquad \frac{\theta_a \sqsubseteq \theta_i \quad (i{=}1 \vee i{=}2)}{\theta_a \sqsubseteq (\theta_1 \vee \theta_2)}\ \textbf{DisjRight}$$

$$\frac{\mathit{fresh}\ y^* \qquad F_0 * D_2 \vdash (\exists\, x^*.\, D_1) * F \qquad F \vdash \theta_a \sqsubseteq \theta_c}{F_0 \vdash (\exists\, x^* \cdot \mathbf{req}\ D_1; \theta_a) \sqsubseteq (\exists\, y^* \cdot \mathbf{req}\ D_2; \theta_c)}\ \textbf{EntReq}$$

$$\frac{\mathit{fresh}\ x^* \qquad F_0 * D_1 \vdash (\exists\, y^*.\, D_2) * F \qquad F \vdash \theta_a \sqsubseteq \theta_c}{F_0 \vdash (\exists\, x^* \cdot \mathbf{ens}[r]\ D_1; \theta_a) \sqsubseteq (\exists\, y^* \cdot \mathbf{ens}[r]\ D_2; \theta_c)}\ \textbf{EntEns}$$

$$\frac{\mathit{fresh}\ x^*, y^* \quad \pi_p{=}\mathit{xpure}(\sigma \wedge \pi) \quad \pi_p \Rightarrow (\exists y^* \cdot x_1^*{=}x_2^* \wedge r_1{=}r_2) \quad emp * \pi_p \vdash \theta_a \sqsubseteq \theta_c}{\sigma \wedge \pi \vdash \big(\exists\, x^* \cdot f(x_1^*, r_1)\big); \theta_a \sqsubseteq \big(\exists\, y^* \cdot f(x_2^*, r_2)\big); \theta_c}\ \textbf{EntFunc}$$

$$\frac{\mathit{xpure}(\sigma) \wedge \pi_1 \Rightarrow \exists\, x^*.\, \pi_2}{\sigma \wedge \pi_1 \vdash (\exists\, x^*.\, emp \wedge \pi_2) * (\sigma \wedge \pi_1)}\ \textbf{SLBase} \qquad \frac{D_1 \vdash (\exists\, x^*.\, D_2 \wedge v_1{=}v_2) * F}{y \mapsto v_1 * D_1 \vdash (\exists\, x^*.\, y \mapsto v_2 * D_2) * F}\ \textbf{SLMatch}$$

$$\frac{\mathit{fresh}\ x \qquad D_1 \wedge z{=}x \vdash (\exists\, y^*.\, D_2 \wedge v_1{=}v_2) * F}{z \mapsto v_1 * D_1 \vdash (\exists\, xy^*.\, x \mapsto v_2 * D_2) * F}\ \textbf{SLMatchEx}$$

Fig. 13: Staged Subsumption and Selected Entailment Rules

abstracted to first-order logic (via the function *xpure*), and the final proof obligation is checked via SMT. We use *xpure* to soundly approximate the spatial information of a heap formula in first-order logic. We illustrate its behavior by example: $xpure(emp) = true$ and $xpure(x \mapsto 1 * y \mapsto 2) = x{\neq}\texttt{null}{\wedge}y{\neq}\texttt{null}{\wedge}x{\neq}y$.

G.2 Inductive predicates and unfolding

The proof system we propose allows inductive predicates, which are useful for specifying data structures (as well) as the behaviors of recursive functions – examples of these were given in Section 2 and later in the appendix [13]. Unlike in classic separation logic, where such predicates are used to describe the shapes of data in the heap, here inductive predicates describe *flows* via staged formulae.

An inductive predicate definition is of the form $g(x^*, r) \triangleq \Phi_g$, where g, x^*, and r may occur in Φ_g. Unfolding an inductive definition simply replaces a function stage $g(y^*, r_g)$ with $[r{:=}r_g][x^*{:=}y^*]\Phi_g$ This is the meat of the **Unfold** rule, which then normalizes the result to Φ_u before continuing.

$$\frac{\begin{array}{c}\exists\, u^* \cdot [r{:=}r_g][x{:=}x_g]\Phi_g; \theta_a \Longleftrightarrow \Phi_u \\ g(x, r) \triangleq \Phi_g \qquad F_0 \vdash \Phi_u \sqsubseteq \big(\exists\, w^* \cdot f(x_f, r_f)\big); \theta_c\end{array}}{F_0 \vdash \big(\exists\, u^* \cdot g(x_g, r_g)\big); \theta_a \sqsubseteq \big(\exists\, w^* \cdot f(x_f, r_f)\big); \theta_c}\ \textbf{Unfold}$$

The first premise in the rule above may be satisfied by a previous definition of an inductive predicate, or by a local name bound to a logical lambda expression. For example, we may unfold y in the flow $\textbf{ens}[\_]\ y{=}\lambda\,(x*, r) \to \Phi; y(v*)$. This underscores the first-class nature of lambda terms in the logic, which is necessary to precisely model higher-order behaviors like currying. Such logical lambda terms are manipulated mostly by the above rule and are encoded in a form that preserves alpha equivalence before being sent to the SMT solver.

One more ingredient is required for reasoning about recursive functions: the use of induction. Inductive predicates provide the means of specification, and annother piece is the automated application of lemmas which is left in the appendix [13].

H Proving Lemmas and Rewriting

Nontrivial proofs in automated program verifiers are often made possible by user-supplied lemmas. A typical way to provide such lemmas in verifiers for Hoare logic and two-state separation logic is to make use of the Curry-Howard correspondence. By encoding the lemma $P \Rightarrow Q$ as a ghost procedure with precondition P and postcondition Q, application of the procedure at a specific point corresponds to applying the lemma to the proof state at that point, and the lemma can be separately proved by writing a body for the procedure.

By analogy to this, we allow *subsumption lemmas* in the system of the form $\forall x^*.\ f(y, r) \sqsubseteq \theta_q$, where x^* may occur free in both $f(y, r)$ and θ_q. This may be seen as a specific case of the subsumption relation between two disjunctive staged formulae, but with singleton disjuncts on both sides, and only a single function stage on the left. The restricted form is sufficient to express the induction hypotheses of subsumptions between recursive functions.

Rewriting is very similar to unfolding – the one difference is that not all arguments of the function stage on the left are parameters, hence the universal quantifier on some.

With both the unfolding and rewriting rules, the proof system is fully defined. As an example, we consider proving a property of following function.

```
let rec applyN f x n =
  °applyN(f, x, n, res) = ···
  if n = 0 then x
  else let r = f x in applyN f r
      (n-1)
```

```
let incr x = x + 1
let summary x n
 summary(x, n, res) =
     req n≥0; ens[_] res=x+n
= applyN incr x n
```

The proof begins with a suitable induction hypothesis. We infer the following one using heuristics. In general, it would have to be provided by the user as a lemma (for re-summarization).

$$\forall\, x, n, res.\ applyN(incr, x, n, res) \sqsubseteq \mathbf{req}\ n{\geq}0\ \mathbf{ens}[\_]\ res{=}x{+}n$$

The following proof can then be carried out automatically. it illustrates the general approach: subsumption proofs go stage by stage, attempting to match function stages by interleaving unfolding and normalization. This ends with either an unknown function stage on both sides or a simple pre- and postcondition, in which case the separation logic proof obligations are discharged, or a pair of differing unknown function stages, in which case the proof fails.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{n \geq 0 \Rightarrow true \qquad res = (x+1)+n-1 \Rightarrow res = x+n}{\boxed{\mathbf{ens}[\_]\ n > 0 \land r = x+1 \land res = r+n-1} \sqsubseteq \mathbf{req}\ n \geq 0; \mathbf{ens}[\_]\ res = x+n}\textsc{EntNorm}}{\mathbf{ens}[\_]\ n{>}0; \boxed{(\mathbf{ens}[\_]\ r{=}x{+}1)}; (\mathbf{req}\ n{-}1 \geq 0; \mathbf{ens}[\_]\ res{=}x{+}n{-}1) \sqsubseteq \mathbf{req}\ n{\geq}0; \mathbf{ens}[\_]\ res{=}x{+}n}\textsc{Normalize}}{\mathbf{ens}[\_]\ n{>}0; inc(x,r); \boxed{(\mathbf{req}\ n{-}1 \geq 0; \mathbf{ens}[\_]\ res{=}x{+}n{-}1)} \sqsubseteq \mathbf{req}\ n{\geq}0; \mathbf{ens}[\_]\ res{=}x{+}n}\textsc{Unfold}}{\boxed{(\ldots \lor \mathbf{ens}[\_]\ n > 0; inc(x,r); applyN(inc, n-1, r, res))} \sqsubseteq \mathbf{req}\ n \geq 0; \mathbf{ens}[\_]\ res = x+n}\textsc{Rewrite}}{applyN(inc, n, x, res) \sqsubseteq \mathbf{req}\ n \geq 0; \mathbf{ens}[\_]\ res = x+n}\textsc{Unfold}$$

In the above example, with $\boxed{\text{boxes}}$ indicating changes from one step to the next, we unfold *applyN* and focus on the inductive case. After rewriting with the induction hypothesis, we unfold *inc* and normalize to compact everything into a single **req**/**ens** stage, allowing an application of **EntNorm** to produce first-order proof obligations.

I Soundness

I.1 Operational semantics

To facilitate the following soundness proofs, we define a big-step reduction relation with judgments of the form $e, h, S \rightsquigarrow R_e, h_1, S_1$. Program states consist of a heap h and store S, like in Section 3.1. Outcomes $R_e ::= Norm(v) \mid Err$ are more constrained compared to R – evaluation results in a value v, not a variable r, and there is only a *Err* outcome representing a failure, without *possible failure* as there might be in the specification.

$$\frac{}{v, h, S \rightsquigarrow v, h, S}\ \textbf{Nil,Const,Lambda} \qquad \frac{}{x_1{::}x_2, h, S \rightsquigarrow S(x_1){::}S(x_2), h, S}\ \textbf{Cons}$$

$$\frac{}{x, h, S \rightsquigarrow S(x), h, S}\ \textbf{Var} \qquad \frac{h, S \models \sigma\wedge\pi}{(assert\ \sigma\wedge\pi), h, S \rightsquigarrow (), h, S}\ \textbf{Assert}$$

$$\frac{e_1, h, S \rightsquigarrow v, h_1, S_1 \qquad e_2, h_1, S_1[x{:=}v] \rightsquigarrow v_1, h_2, S_2}{(let\ x{=}e_1\ in\ e_2), h, S \rightsquigarrow v_1, h_2, S_2[x{:}{\not=}]}\ \textbf{Let}$$

$$\frac{S(f) = fun\,(x^*)\,\Phi{\rightarrow}\,e \qquad S(x^*) = v^* \quad [x^*{:=}v^*]e, h, S \rightsquigarrow v_2, h_1, S_1}{f(x^*), h, S \rightsquigarrow v_2, h_1, S_1}\ \textbf{App} \qquad \frac{x_1 \in dom(h)}{(x_1 := x_2), h, S \rightsquigarrow (), h_1[S(x){:=}S(v)], S}\ \textbf{Assign}$$

$$\frac{}{!x, h, S \rightsquigarrow h(S(x)), h, S}\ \textbf{Deref} \qquad \frac{l \notin dom(h)}{ref\ x, h, S \rightsquigarrow \ell, h[\ell{:=}S(x)], S}\ \textbf{Ref}$$

$$\frac{S(b){=}true \qquad e_1, h, s \rightsquigarrow v, h_1, S_1}{(if\ b\ then\ e_1\ else\ e_2), h, S \rightsquigarrow v, h_1, S_1}\ \textbf{If1} \qquad \frac{S(b){=}false \qquad e_2, h, s \rightsquigarrow v, h_1, S_1}{(if\ b\ then\ e_1\ else\ e_2), h, S \rightsquigarrow v, h_1, S_1}\ \textbf{If2}$$

I.2 Soundness of Normalization

Theorem 1 (Soundness of Normalization). *Given* $\Phi_1{\Longrightarrow}\Phi_2$*, if* $S, H \rightsquigarrow S_1, H_1, R_1 \models \Phi_1$*, then* $S, H \rightsquigarrow S_1, H_1, R_1 \models \Phi_2$*.*

Intuitively, normalization of a staged formula preserves its behavior.

Proof. By case analysis on the derivation of $\Phi_1{\Longrightarrow}\Phi_2$. Most cases follow immediately from the semantics of staged formulae (Fig. 5). For example, given ${\Longrightarrow}\textbf{req}\ D_1; \textbf{req}\ D_2 \textbf{req}\ D_1 * \textbf{req}\ D_2$, $h = h_0 \circ h_1 \circ h_2$, where $h_1 \models D_1$ and $h_2 \models D_2$, and $L_1 = h_1 \circ h_2$.

The only remaining case is **Float Pre** (Section 3.2). Here we reason about heaps instead of heap formulae. Let $h(D_i)$ be a heap such that $h(D_i) \models D_i$ for $i \in \{a, f, 1, 2\}$. First, it must be the case that D_a describes a heap that is in h, otherwise it would be impossible to prove the conclusion. From this we know $h = h(D_a) \circ F$ for some framing heap F. Now, from the second premise, we have $h \circ h(D_1) = h(D_2) \circ h_1$. Substituting h, we have $(h(D_a) \circ F) \circ h(D_1) = h(D_2) \circ h_1$. From the first premise, assuming soundness of $\vdash$, we have $h(D_a) \circ h(D_1) = h(D_2) \circ h(D_f)$. h_1 must thus be $h(D_f) \circ F$. Consider the conclusion. It suffices to prove that $h = h(D_a) \circ F$ and $h(D_f) \circ F = h_1$, which is exactly what we have.

I.3 Soundness of Forward Rules

The rules are sound if every derivable triple $\{\ \mathit{emp}\ \}\ e\ \{\ \Phi\ \}$ is valid.

A staged formula Φ is valid if: given that it specifies a transition from initial configuration S, h to final configuration $S_2, h_1, \mathit{Norm}(r)$, execution of e from the same initial configuration results in a *compatible* final configuration $h_1, S_1, \mathit{Norm}(v)$.

Compatibility requires that $S_1 \subseteq S_2$ and $S_1(r) = v$. Intuitively, the heap and result have to be equal, while the final store of Φ is allowed to contain more bindings due to existentials.

Theorem 1 (Soundness of Forward Rules) *Given* $\{\ \mathit{emp}\ \}\ e\ \{\ \Phi\ \}$*, then* $\forall S, h, S_2, h_1 \cdot (S, h \rightsquigarrow S_2, h_1, \mathit{Norm}(r) \models \Phi) \Rightarrow \exists S_1 \cdot e, h, S \rightsquigarrow \mathit{Norm}(v), h_1, S_1$ *and* $S_1 \subseteq S_2$ *and* $S_1(r) = v$*.*

Proof. By induction on the derivation of $e, h, S_1 \rightsquigarrow \mathit{Norm}(v), h_1, S_1$.

- The nil, cons, and constant are straightforward: their reduction has no effect on the heap, Φ is of the form $\mathbf{ens}[\mathit{res}]\ \mathit{res}{=}v$, $S_1 = S_2$, and $S_1(\mathit{res}) = v$.
- The variable case is similar: Φ is of the form $\mathbf{ens}[\mathit{res}]\ \mathit{res}{=}x$, $S_1 = S_2$, and $S_1(\mathit{res}) = x$.
- The lambda case is immediate from the induction hypothesis on e, assuming that entailment is sound.
- e is of the form $\mathit{ref}\ x$. Reduction results in a store $S_1 = S$ and heap $h_1 = h[\ell{:=}S(x)]$. Φ is of the form $\mathbf{ens}[\mathit{res}]\ \mathit{res} \mapsto x$, with final state $S_2 = S$ and heap h_1, if we choose $h_1 = \{\ell \mapsto S(x)\}$. We also have $S_1(\mathit{res}) = v = \ell$.
- e is of the form $x_1 := x_2$. Reduction results in a store $S_1 = S$ and heap $h_1 = h[S(x_1){:=}S(x_2)]$. Given Φ is of the form $\mathbf{req}\ x_1 \mapsto \_; \mathbf{ens}[\_]\ x_1 \mapsto x_2$, reduction removes the heap location x_1, then re-adds it via with new value $S(x_2)$, effectively modifying the heap at that the assigned location to h_1.
- e is of the form $!x$. Reduction has no effect on the heap and store and results in $v = S(x)$. Φ is of the form $\exists\, a \cdot \mathbf{req}\ x \mapsto a; \mathbf{ens}[\mathit{res}]\ x \mapsto a \wedge \mathit{res}{=}a$, reduction removes some heap location ℓ with value v, then immediately readds it, leaving the heap unchanged. The store S_2 is extended with $\{a \mapsto v\}$. It is a superset of S_1 because $h(S(x)) = h(\ell) = v$.
- e is of the form *assert* D. Reduction has no effect on the heap, and reduction of Φ is similar to the case for $!x$.
- e is of the form *let* $x{=}e_1$ *in* e_2. Reduction evaluates e_1 to a value v, evaluates e_2 with x bound to v in the store, then removes the binding for x, resulting in a heap h and store S_1. Φ is $\Phi = \exists\, x \cdot [\mathit{res}{:=}x]\Phi_1; [x{:=}x]\Phi_2$, where $\{\ \mathit{emp}\ \}\ e_1\ \{\ \Phi_1\ \}$ and $\{\ \mathit{emp}\ \}\ e_2\ \{\ \Phi_2\ \}$. Given the induction hypotheses, it remains to show that the heap resulting from reducing Φ is h and the resulting store S_2 is a superset of S_1. The former follows directly from the semantics of staged formulae as *let* expressions do not further change the heap.
- e is of the form *if* x *then* e_1 *else* e_2. Reduction proceeds with either e_1 or e_2, depending on the value of $S(x)$. The conclusion follows from case analysis on x, then application of either induction hypothesis.

– e is of the form $f(x^*)$. Its a summary is a simple function stage $f(x^*, res)$. From the big-step reduction relation, we know that e evaluates as $[x^*{:=}S(x^*)]e_1$ would, given $S(f) = \mathit{fun}\,(x^*)\,\Phi \rightarrow e_1$. Also, from the semantics of staged formulae, it suffices to prove soundness for $[y^*{:=}x^*]\Phi$. We know that e_1 is summarized soundly from the induction hypothesis, and the substitutions do not compromise this as both e_1 and Φ contain (the values of) x^* after.

I.4 Soundness of Entailment

As we use a standard fragment of separation logic with the usual semantics, we assume the soundness of its entailment proof system.

Theorem 3 (Soundness of SL entailment). *Given $D_1 \vdash D_2$ and $S, h \models D_1$ then $S, h \models D_2$.*

We focus on the soundness of the entailment proof system for staged formulae. Given a derivation $\Phi_1 \sqsubseteq \Phi_2$ and the same starting configuration S, H with an empty local heap, "executing" Φ_1 should result in an an ending configuration with a smaller global heap and larger local heap than executing Φ_1. The relation between heaps is containment rather than equality intuitively because a stronger formula is allowed to entail a weaker one, e.g., $\mathbf{ens}[\_]\ x \mapsto 1 * y \mapsto 1$ should entail $\mathbf{ens}[\_]\ x \mapsto 1$.

Theorem 2 (Soundness of Entailment). *Given $\Phi_1 \sqsubseteq \Phi_2$ and $S, h \rightsquigarrow \mathit{Norm}(r_1), S_1, h_1 \models \Phi_1$, then there exists h_2 such that $S, h \rightsquigarrow \mathit{Norm}(r_1), S_2, h_2 \models \Phi_2$ where $h_2 \subseteq h_1$.* (Here, $h_1 \subseteq h_2$ denotes that $\exists\, h_3.\, h_1 \circ h_3 = h_2$.)

Proof. By induction on the derivation of $\Phi_1 \sqsubseteq \Phi_2$.

– In **DisjLeft**, when the derivation is of the form $(\Phi_1 \vee \Phi_2) \sqsubseteq \Phi_2$, we know from the induction hypothesis that execution of both Φ_1 and Φ_2 result in global heaps H_3 and H_4 that are larger than H_2, and local heaps L_3 and L_4 that are smaller than L_2. Since both ending configurations satisfy the property, and the semantics of disjunction allows us to choose either ending configuration, the conclusion is true.
– In **DisjRight**, when the derivation is of the form $\Phi_a \sqsubseteq (\Phi_1 \vee \Phi_2)$, we know from the induction hypothesis that only the execution of Φ_1 produces an ending configuration that satisfies the property. Since the semantics of disjunction allow execution of $\Phi_1 \vee \Phi_2$ to choose either Φ_1 or Φ_2, it suffices to prove the property for just one disjunct, which we have by the induction hypothesis.
– In **EntReq**, when the derivation is of the form $F_0 \vdash (\exists\, x^* \cdot \mathbf{req}\ D_1; \Phi_a) \sqsubseteq (\exists\, y^* \cdot \mathbf{req}\ D_2; \Phi_c)$, we must show that $\mathbf{req}\ D_2$ results in an identical store, smaller global heap, and larger local heap than $\mathbf{req}\ D_1$ (and the induction hypothesis ensures that these relations are preserved). In other words, D_2 should move *more* of the global heap than D_1 does into L_2 (resp. L_1). Since we know $\mathit{emp} * D_2 \vdash D1$, from Theorem 3, we know that any heap satisfying D_2 must satisfy D_1. D_2 must thus be a more precise heap formula, with

more conjuncts. Thus, it may removed more of the heap than D_1, making H_2 smaller. Correspondingly, L_2 will be larger than L_1 as a larger portion of the heap was moved into it. **req** does not update the store, hence the conclusion is true.
- The **EntEns** case is dual to **EntReq**, but largely similar. Given a derivation of the form $F_0 \vdash (\exists x^* \cdot \mathbf{ens}[\_]\ D_1; \Phi_a) \sqsubseteq (\exists y^* \cdot \mathbf{ens}[\_]\ D_2; \Phi_c)$, we know $D_1 \vdash D_2$. Hence D_1 must add a larger portion of heap into the global heap H_1 than D_2 does, resulting in H_1 being larger and L_1 being smaller. **ens** does update *res* in the store, but the compatibility of the values that the store is updated with is ensured by the validity of the separation logic entailment.
- The **EntFunc** case seeks to prove that both sides of the entailment are effectively equal under the assumptions in F_0. As the store and both heaps are modified the same way (by the formula that is the definition of f), they remain equal, hence the conclusion is true.
- Both rules for unfolding are technically involved, but conceptually simple. As we replace function stages with their specifications (which have been earlier checked for soundness), the conclusion follows from the soundness of normalization (Theorem 1).

I.5 Termination

We outline termination arguments for our normalization procedure, Hoare-style forward rules and also the entailment/subsumption procedure for staged specifications

I.6 Termination of Normalization

The normalization procedure $\Phi_1 \Longrightarrow \Phi_2$ is terminating since staged logic expression is always getting smaller, except for the *FloatPre* rule, where each pre-condition is being floated outwards. Hence, by giving lower weights to pre-condition stages, this rule will continues to decrease the termination measure of our normalization procedure.Hence, each application of normalization/compaction will always terminate.

I.7 Termination of Hoare-Style Forward Rules

Except for the consequence rule, all the forward reasoning rules are either just terminating (CVar, Val, Assign, Call and Assert). or works on structurally smaller expressions (e.g. Lambda, Let and If Hoare-rules). If we assume that Consequence rule is applied at most once for each distinct sub-expression, the set of Hoare-style forward-reasoning rules is always terminating.

I.8 Termination of Staged Entailment

Staged Entailment is used mostly for the summarization of recursive staged predicates. This process requires users to provide suitable lemmas that must be appropriately generalized. In case this is not done properly, there is a possibility that staged entailment may go into an infinite loop. To prevent this problem, we only allow a bounded set of new staged predicate definitions to be defined, and a bound number of unfoldings. Once these two bounds are exceeeded, we approximate our staged entailment outcome with a failure. This approximation is sound but may lead to incompleteness for our entailment procedure over staged logics.