Chapter 1
Automated Verification with HIP/SLEEK and Second-Order Abductive Inference

Wei-Ngan Chin, Andreea Costea, Quang Loc Le, Ton Chanh Le, Benedict Lee, Thanh-Toan Nguyen, and Quang-Trung Ta

Abstract There are several important factors to consider for automated program verification systems, including modularity, precision, expressivity and automation. The HIP/SLEEK verification system was designed with a bunch of these factors in mind. We were inspired by a new logic (at that time), known as separation logic, which forms the core of our specification logic. From the beginning, we value program proving as much as program analysis; and would focus first on the development of expressive specification and automated verification, before program analyses are subsequently considered in an orthogonal fashion. This approach allows us to experiment with specification and proving, before using them to further guide the inference process. We have also integrated both proving and inference within the same entailment system with the help of second-order biabductive reasoning. While this approach takes longer to develop, we believe it is a more principled and systematic way to develop automated program verification systems.

Key words: Software verification, Program analysis, Competition on Software Verification, Separation Logic

---

Wei-Ngan Chin
School of Computing, National University of Singapore

Andreea Costea
EEMCS, TU Delft

Quang Loc Le
Department of Computer Science, University College London

Ton Chanh Le
Algorand

Benedict Lee
GovTech Singapore

Thanh-Toan Nguyen
School of Computing, National University of Singapore

Quang-Trung Ta
School of Computing, National University of Singapore

1.1 Introduction

The HIP/SLEEK project is an early adopter and pioneer in the use of separation logic to support reasoning for heap-manipulating programs. Separation logic allows must-aliasing where strong updates can be supported for mutable heap nodes, and the use of inductive predicates can capture data structures with complex invariant properties, such as sorted lists or near-balanced AVL trees.

In the course of the project, we have focused on several major issues of automated program verification, namely *precision*, *modularity*, *expressivity* and *automation*.

Precision is supported by must-aliasing from pointer-based logic, together with suitable addition of pure logic, such as arithmetic or set properties [68].

Modularity is handled by our focus on a per method (or loop) verification via Hoare-style pre/post specifications.

Expressivity is supported by the use of structured specifications (with support for proof search, case analysis and staged verification) [33] and the use of lemmas (relating arbitrary predicates) that can be automatically applied and proven [67].

Lastly, proof automation is supported by an algorithm that is built on top of a set of sound proof rules. In our project, the proof system progressively matches the antecedent and consequent of each proof entailment, until some residue form is discovered. Both frame and anti-frame can be captured in our automated entailment prover, to help support both frame and abductive inferences. Moreover, we support proof search with the help of a set of outcomes from the entailment proving process. These designs allow us to support both automated proof and modular program analyses, allowing a smoother transition from proving to program analysis, in this new extended specification logic.

Our HIP/SLEEK project has also progressed beyond functional correctness. We have an extension that supports both temporary and permanent immutability [26], and another extension which supports both use-site and field immutability. We have also designed an extended resource logic to help us reason about both termination as well as non-termination. Both these properties can be captured by our enhanced logic with resource specification support that mirrors upper and lower bound of computation. These can moreover be discovered by specification inference mechanisms through second-order abductive reasoning.

1.2 HIP – Verification System

We present an overview of our verification system HIP/SLEEK in Fig. 1.1. Its front-end is a Hoare-Floyd style forward-verifier, called HIP, which analyzes C-like imperative programs against their specifications written in pairs of pre- and post-conditions. Its back-end is a logical prover, called SLEEK, which examines verification conditions generated by HIP to decide whether the input programs satisfy their specifications. The details of this verification process are as follows:

- Firstly, the verifier HIP takes as input the C-like source code of the program to be verified along with its pre-/post-specifications;

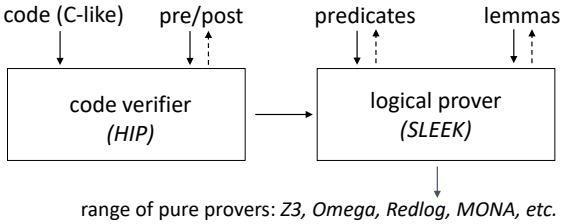

Fig. 1.1 HIP/SLEEK overview: HIP checks whether the code verifies against its attached pre-/post-conditions, discharging proof obligations to SLEEK. In its turn, SLEEK relies on a number of off-the-shelf constraint solvers and theorem provers. The solid arrows show the workflow of the verification, and the dashed arrows are used to denote inference of predicates, lemmas and/or pre-/post-specifications).

- Then, HIP verifies the input code against its specification in two phases. (i) For each method definition, HIP assumes that the pre-condition holds, and it derives the strongest post-condition upon the method's body and checks whether the derived post-condition entails the declared post-condition (see METH–DEF in Sub. 1.2.3). (ii) For each method call, HIP checks whether the callee's pre-condition holds at the caller's site. If yes, it adds the corresponding post-condition to the caller's abstract state (see METH–CALL in Sub. 1.2.3). See Fig. 1.4 for a full list of the forward verification rules.
- The verification rules often involve discharging proof obligations in the form of entailment checks, solved by the prover SLEEK (detailed in Sec. 1.3). SLEEK reasons about user-defined predicates, and, consequently, lemmas which relate the different predicates describing the same resource.
- The specification language supported by HIP/SLEEK is rich (see Fig. 1.3 for the full specification language), comprising not only shape properties, but also numerical ones, such as size, values, sortedness, etc. To handle all these theories, our system interacts with a series of off-the-shelf constraint solvers and theorem provers, such as Z3 [66], Omega Calculator [81], MONA [28], etc.

1.2.1 The Programming Language

A program $\mathcal{P}$ (Fig. 1.2) contains user-defined data structures (*data*) and methods (*method*). Each method is decorated with pre-/post-specifications describing the program's functionality and safety conditions. A method can take both pass-by-reference (`ref t v`) parameters as well as pass-by-value ones (`t v`). Memory is allocated using the `new` operator, and `v.f`. is used to dereference data structures fields. The underlying language also supports loops (decorated with loop invariants) but treats them as tail-recursive calls (and translates the loop invariants into sets of pre-/post-specifications).

Data structures are defined in HIP/SLEEK as follows:

```
data node { int val; node next;}
data tree { int val; tree left; tree right;}
```

A program which recursively appends two linked-lists may be written as follows:

(Program)	$\mathcal{P}$	::= $data^*$ $method^*$
(Data Struct.)	$data$	::= `struct` d { $(t\,f)^*$ }
(Method Def.)	$method$	::= t mn((`ref` t v)*, (t v)*)(`requires` Φ_{pr} `ensures` Φ_{po};)*{`e`}
(Types)	t	::= `d` $\mid$ τ
(Primitive Types)	τ	::= `int` $\mid$ `bool` $\mid$ `float` $\mid$ `void`
(Expressions)	`e`	::= `NULL` $\mid$ $\mathtt{k}^{\tau}$ $\mid$ `v` $\mid$ `new` `d`(`v`*) $\mid$ `t` `v`; `e` $\mid$ `v.f` $\mid$ mn(`v`*)
		$\mid$ `v:=e` $\mid$ `v.f:=e` $\mid$ `e`;`e` $\mid$ `if` (`b`) `e` `else` `e` $\mid$ `return` `e`
(Boolean Expr.)	`b`	::= `e==e` $\mid$ `!(b)` $\mid$ `b&b` $\mid$ `b|b`

where $\mathtt{k}^{\tau}$ is a constant of type τ, `v` is a variable, `f` denotes a field, and mn is a method's name.

Fig. 1.2 A Core Imperative Language.

```
1       void append(node x, node y)
2       {
3         if (x.next==NULL) x.next = y;
4         else {
5           // x.val = 0;
6           append(x.next,y);
7         }
8       }
```

Although HIP/SLEEK can verify concurrent programs too, this paper only describes the verification of programs in the sequential setting. Details on the extension of HIP/SLEEK with a session logic to verify communication centered programs are found at [23, 22, 20], while details on the verification of barriers and other synchronization mechanisms are found at [49, 50, 51, 86]. If the reader is further interested in how to use HIP/SLEEK for program repair, [61, 69, 70] details that too.

1.2.2 The Specification Language

The specification language (see Fig. 1.3) is built on an extension of Separation Logic [83, 39] with support for separation conjunction $*$ and inductive predicates. The spatial formula $\kappa_1 * \kappa_2$ asserts that the heap can be split into two disjoint parts in which κ_1 and κ_2 hold, respectively. Moreover, $\mathtt{u}{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}\rangle$ is a singleton heap (or heaplet) modelling a single data structure having `u` as its root address and $\overline{\mathtt{v}}$ as values of its fields. For example, the formula $\mathtt{u}{\mapsto}\mathtt{node}\langle 0, \mathtt{v}\rangle$ represents a data structure of type `node` whose first field (`val`) has the value 0, and the second field (`next`) is a pointer `v`. Furthermore, $\mathtt{P}(\overline{\mathtt{v}})$ is an inductive heap modelling a recursive data structure. In our work, by convention, the first argument in $\overline{\mathtt{v}}$ is the "root" pointer to the specified data structure that guides the data traversal in $\mathtt{P}(\overline{\mathtt{v}})$. Besides the spatial and first order logic operators, the specification language also allows the user to write arithmetic and bag constraints.

The descriptions of the data structures come in the form of inductive predicates which capture the shape and numerical properties of the underlying data structures. Since

(Symbolic pred.)	*pred*	$::= \mathtt{P}(\overline{\mathtt{v}}) \triangleq \Phi$ inv π
(Formula)	Φ	$::= \bigvee \Delta \qquad \Delta ::= \exists\, \overline{\mathtt{v}} \cdot \kappa \wedge \pi \mid \Delta * \Delta$
(Spatial formula)	κ	$::= \mathtt{emp} \mid \mathtt{v} \mapsto \mathtt{d}\langle \overline{\mathtt{v}} \rangle \mid \mathtt{P}(\overline{\mathtt{v}}) \mid \kappa * \kappa \mid \mathtt{V}$
(Pure)	π	$::= \mathtt{b} \mid \mathtt{s} \mid \pi \wedge \pi \mid \pi \vee \pi \mid \neg \pi \mid \exists \mathtt{v} \cdot \pi \mid \forall \mathtt{v} \cdot \pi \mid \gamma$
(Pointer eq./diseq.)	γ	$::= \mathtt{v}{=}\mathtt{v} \mid \mathtt{v}{=}\mathtt{null} \mid \mathtt{v} \neq \mathtt{v} \mid \mathtt{v} \neq \mathtt{null}$
(Boolean)	b	$::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{b}{=}\mathtt{b} \qquad \mathtt{s} ::= \mathtt{s}{=}\mathtt{s} \mid \mathtt{s} \leq \mathtt{s} \mid \mathtt{V}{=}\Delta$
(Presburger Arith.)	s	$::= \mathtt{k}^{\tau} \mid \mathtt{v} \mid \mathtt{k}^{\tau} \times \mathtt{s} \mid \mathtt{s}{+}\mathtt{s} \mid -\mathtt{s}$
(Bag Constraint)	φ	$::= \mathtt{v} \in \mathcal{B} \mid \mathcal{B}{=}\mathcal{B} \mid \mathcal{B} \sqsubset \mathcal{B} \mid \forall \mathtt{v} \in \mathcal{B} \cdot \pi \mid \exists \mathtt{v} \in \mathcal{B} \cdot \pi$
	$\mathcal{B}$	$::= \mathcal{B} \sqcup \mathcal{B} \mid \mathcal{B} \sqcap \mathcal{B} \mid \mathcal{B} - \mathcal{B} \mid \emptyset \mid \{\mathtt{v}\}$

where $\mathtt{k}^{\tau}$: constant of type τ; v : first order variable;
V : second-order variable;
d : name of a user-defined data structure

Fig. 1.3 The Specification Language

we do not restrict what data structures the program uses, we also do not restrict their corresponding predicates – thus the user has the freedom to write the necessary predicates as they obey some well-foundness conditions (detailed later in the section). For example, the predicates below inductively describe **the shape** of a linked-list and a linked-list segment, respectively:

$$\mathtt{ll}(\mathtt{x}) \triangleq \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{null} \vee \exists \mathtt{q} \cdot (\mathtt{x} \mapsto \mathtt{node}\langle \_, \mathtt{q} \rangle * \mathtt{ll}(\mathtt{q})).$$
$$\mathtt{lseg}(\mathtt{x}, \mathtt{p}) \triangleq \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{p} \vee \exists \mathtt{q} \cdot (\mathtt{x} \mapsto \mathtt{node}\langle \_, \mathtt{q} \rangle * \mathtt{lseg}(\mathtt{q}, \mathtt{p})).$$

The `ll` (or `lseg`) predicate asserts that a linked-list (or a linked-list segment) can be *empty* when the root pointer to the structure is `null`, `x=null` (or equal to the pivot node `x=p`), or *non-empty* when the root pointer refers to a `node` whose `next` field points to a linked-list (or linked-list segment, respectively). The separating conjunction in the inductive case ensures that the head and the tail of the linked-list structure reside in disjoint heaps. We use underscore (_) in the formulae to denote an existentially quantified (anonymous) variable. All the non-parameter variables in the definition of a predicate are existentially quantified, even when not explicitly stated as so. The specification for the `append` code given in the previous subsection, may now be expressed as:

```
requires ll(x) * ll(y) ∧ x≠null
ensures  ll(x);
```

However, even though this specification is correct, it is actually too weak to capture the gist of the append operation. Instead we could write the following more precise specification where the pre-condition on the `y` parameter is a weaker list segment (instead of null-terminated `ll(y)` list):

```
requires lseg(x,null) * lseg(y,q) ∧ x≠null
ensures  lseg(x,q);
```

These two definitions (ll and lseg) can be refined into capturing more precise information about the underlying structures, such as **the size** of the lists:

$$\begin{array}{rl} \mathtt{ll_n(x,n)} & \triangleq (\mathtt{emp} \wedge \mathtt{x{=}null} \wedge \mathtt{n{=}0}) \vee \exists \mathtt{q} \cdot (\mathtt{x} \mapsto \mathtt{node}\langle \_, \mathtt{q}\rangle * \mathtt{ll_n(q,n{-}1)}) \\ & \quad \mathtt{inv\ n{\geq}0}. \\ \mathtt{lseg_n(x,p,n)} & \triangleq (\mathtt{emp} \wedge \mathtt{x{=}p} \wedge \mathtt{n{=}0}) \vee \exists \mathtt{q} \cdot (\mathtt{x} \mapsto \mathtt{node}\langle \_, \mathtt{q}\rangle * \mathtt{lseg_n(q,p,n{-}1)}) \\ & \quad \mathtt{inv\ n{\geq}0}. \end{array}$$

The above definitions also specify a default invariant $\mathtt{n{\geq}0}$ which holds for all $\mathtt{ll_n}$ and $\mathtt{lseg_n}$ list predicates. This predicate invariant can be verified by checking that each disjunctive branch of the predicate definition always implies its stated invariant. Types need not be given in our specification as we have an inference algorithm to automatically infer non-empty types for specifications that are well-typed. For the above predicates, our type inference can determine that n is of int type, while x, q and p are of node type. The specification for the append method could now capture even more information with this improved definitions:

$$\begin{array}{l} \mathtt{requires\ ll_n(x,n) * ll_n(y,m) \wedge x{\neq}null} \\ \mathtt{ensures\ \ ll_n(x,n{+}m);} \\ \mathtt{requires\ lseg_n(x,null,n) * lseg_n(y,q,m) \wedge x{\neq}null} \\ \mathtt{ensures\ \ lseg_n(x,q,n{+}m);} \end{array}$$

The append method verifies against these specifications. But let us consider a scenario for the append method given earlier where the instruction at line 5 is uncommented. That is, the bogus statement x.val = 0 is now assumed to be a part of the append's method body. So the considered method now also resets the values stored in the list segment pointed by x. To avoid this bogus behavior we could refine further the definition of a list segment and the pre-/post-specifications as follows:

$$\begin{array}{rl} \mathtt{lseg_B(x,p,n,B)} & \triangleq (\mathtt{emp} \wedge \mathtt{x{=}p} \wedge \mathtt{n{=}0} \wedge \mathtt{B{=}\emptyset}) \vee \\ & \quad \exists \mathtt{q,v} \cdot (\mathtt{x} \mapsto \mathtt{node}\langle \mathtt{v,q}\rangle * \mathtt{lseg_B(q,p,n{-}1,B_1)} \wedge \mathtt{B{=}B_1 \cup \{v\}}) \\ & \quad \mathtt{inv\ n{\geq}0}. \end{array}$$

$$\begin{array}{l} \mathtt{requires\ lseg_B(x,null,n,B_x) * lseg_B(y,q,m,B_y) \wedge x{\neq}null} \\ \mathtt{ensures\ \ lseg_B(x,q,n{+}m,B_x \cup B_y);} \end{array}$$

which besides the shape and size properties, it also captures **the values** stored in the linked-lists. Going even further with these refinements, one could also define the **sortedness** property of the linked-list segment:

$$\begin{array}{rl} \mathtt{lsort(x,p,n,B)} & \triangleq (\mathtt{emp} \wedge \mathtt{x{=}p} \wedge \mathtt{n{=}0} \wedge \mathtt{B{=}\emptyset}) \vee \\ & \quad \exists \mathtt{q,v} \cdot (\mathtt{x} \mapsto \mathtt{node}\langle \mathtt{v,q}\rangle * \mathtt{lsort(q,p,n{-}1,B_1)} \wedge \mathtt{B{=}B_1 \cup \{v\}}) \\ & \quad \wedge \forall \mathtt{w} \cdot (\mathtt{w} \notin \mathtt{B_1} \vee \mathtt{v{\leq}w}) \\ & \quad \mathtt{inv\ n{\geq}0}. \end{array}$$

The program's specification in this case can be written as:

$$\texttt{requires}\ \texttt{lsort}(\texttt{x},\texttt{null},\texttt{n},\texttt{B}_\texttt{x}) * \texttt{lsort}(\texttt{y},\texttt{q},\texttt{m},\texttt{B}_\texttt{y}) \wedge \texttt{x}{\neq}\texttt{null} \wedge \forall \texttt{v}_\texttt{x},\texttt{v}_\texttt{y} \cdot ((\texttt{v}_\texttt{x} \in \texttt{B}_\texttt{x} \wedge \texttt{v}_\texttt{y} \in \texttt{B}_\texttt{y}) \vee (\texttt{v}_\texttt{x}{\leq}\texttt{v}_\texttt{y}))$$
$$\texttt{ensures}\ \ \texttt{lsort}(\texttt{x},\texttt{q},\texttt{n+m},\texttt{B}_\texttt{x} \cup \texttt{B}_\texttt{y});$$

which ensures that if the two lists given as input are sorted, and, moreover, all the elements in the list pointed by x are less than or equal to the elements in the list segment pointed by y, then the resulting list segment is still sorted.

Multiple Specifications. Each method may be decorated with multiple pairs of pre-/post-conditions. If that is the case, a method definition needs to verify against every pair of associated specifications, while a callee needs to entail at least one pre-condition associated with the corresponding method call (or at least one pre-conditions corresponding to each context of the method call if the same call statement needs to be verified from different contexts). In other words, the caller may choose the pair of specification(s) which caters best for its calling environment. For the append method, all of the specifications provided so far verify against the method's body, hence a user may very well attach all of the earlier specifications to append. The technical details behind this enhancement are found in [15].

Immutability Specifications. It is often the case that a method only mutates some parts of its memory footprint. To capture this fact in the specification, HIP supports a logic of immutability annotations, where data structures are annotated as *mutable* – @M, or *immutable* – @L according to the case. We define a subsumption relation between these annotations, where @M<:@L, meaning that a heap which is annotated as mutable may be both read or mutated, while an immutable heap may only be used for reading operations. There are a number of benefits from using immutability annotations, such as concise specifications, or increased expressivity as compared to the specifications with no immutability annotations. There is an exhaustive list of these benefits and the technical details behind the immutability annotations in [26]. Furthermore, it is also the case that a method may only operate on certain fields of a given data structure. This fact may be described using immutability annotations at the field level rather than at the data structure level, e.g. a list should maintain its shape intact while being passed to a method that only resets its values. The details of how HIP achieves this are found in [21].

Coming back to the append method, one could now write a more compact specification which captures the immutability of both the shape and the values of the list segment pointed by y as follows:

$$\texttt{requires}\ \texttt{lseg}(\texttt{x},\texttt{null}) * \texttt{lseg}(\texttt{y},\texttt{q})@\texttt{L} \wedge \texttt{x}{\neq}\texttt{null}$$
$$\texttt{ensures}\ \ \texttt{lseg}(\texttt{x},\texttt{q});$$

where lseg(y, q) was marked as immutable. As its properties need not be proved, and are just assumed to hold, the list segment is omitted from the post-condition.

Structured Specifications. We show in [33] how adding structure to a flat specification leads to better expressiveness and verifiability. In particular, structured specifications allow a verifier to perform a case analysis in order to take advantage of the disjointness conditions in the logic.

Specifications with the Explicit Notion of Infinity. We show in [87] the benefits of adding the notion of infinity to the specification language. In particular, the specifications are not only more expressive, but also more human readable and more concise, leading to better composability. For example, the sortedness property of linked-list segments may also be captured via the minimum value property when one chooses to avoid the expensive reasoning over bags of values:

$$\begin{array}{l}\mathtt{lsort}_{\mathtt{min}}(\mathtt{x},\mathtt{p},\mathtt{mn}) \triangleq (\mathtt{x}{\mapsto}\mathtt{node}\langle \mathtt{mn},\mathtt{p}\rangle) \vee \\ \qquad\qquad \exists \mathtt{q},\mathtt{mn}_1 \cdot (\mathtt{x}{\mapsto}\mathtt{node}\langle \mathtt{mn},\mathtt{q}\rangle * \mathtt{lsort}_{\mathtt{min}}(\mathtt{q},\mathtt{p},\mathtt{mn}_1) \wedge \mathtt{mn}{\leq}\mathtt{mn}_1).\end{array}$$

but this definition forces the list segment to be non-empty. To support empty case too and to avoid disjunctive specifications which should have dedicated assertions for the empty case, one would write the following definition using the special ghost variable ∞:

$$\begin{array}{l}\mathtt{lsort}^{\infty}_{\mathtt{min}}(\mathtt{x},\mathtt{p},\mathtt{mn}) \triangleq \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{p} \ \wedge \mathtt{mn}{=}\infty \vee \\ \qquad\qquad \exists \mathtt{q},\mathtt{mn}_1 \cdot (\mathtt{x}{\mapsto}\mathtt{node}\langle \mathtt{mn},\mathtt{q}\rangle * \mathtt{lsort}^{\infty}_{\mathtt{min}}(\mathtt{q},\mathtt{p},\mathtt{mn}_1) \wedge \mathtt{mn}{\leq}\mathtt{mn}_1).\end{array}$$

Well-formedness. As highlighted before, separation formulae are used in pre/post-conditions and shape definitions. In order to handle them correctly without running into unmatched residual heap nodes, we require each separation constraint to be well-formed, as defined below:

Definition 1 (Accessible) A variable is accessible if it is a method parameter, or if it is a dedicated variable, such as `res`.

Definition 2 (Reachable) Given a heap constraint κ and a pointer constraint γ, the heap nodes in κ that are reachable from a set of pointers S can be computed by the following recursively defined function:

$$\begin{array}{l}\mathtt{reach}(\kappa,\gamma,\mathtt{S}) \triangleq \mathtt{u}{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}\rangle * \mathtt{reach}(\kappa - (\mathtt{u}{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}\rangle),\gamma,\mathtt{S}\cup\{\mathtt{v}|\mathtt{v}{\in}\overline{\mathtt{v}},\mathtt{IsPtr}(\mathtt{v})\}) \\ \qquad\qquad \text{when } \exists \mathtt{q}{\in}\mathtt{S}\cdot(\gamma \implies \mathtt{u}{=}\mathtt{q}) \wedge \mathtt{u}{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}\rangle \in \kappa \\ \mathtt{reach}(\kappa,\gamma,\mathtt{S}) \triangleq \mathtt{P}(\overline{\mathtt{v}}) * \mathtt{reach}(\kappa - (\mathtt{P}(\overline{\mathtt{v}})),\gamma,\mathtt{S}\cup\{\mathtt{v}|\mathtt{v}{\in}\overline{\mathtt{v}},\mathtt{IsPtr}(\mathtt{v})\}) \\ \qquad\qquad \text{when } \exists \mathtt{q}{\in}\mathtt{S},\mathtt{p}{\in}\overline{\mathtt{v}}\cdot(\gamma \implies \mathtt{p}{=}\mathtt{q}) \wedge \mathtt{P}(\overline{\mathtt{v}}) \in \kappa \\ \mathtt{reach}(\kappa,\gamma,\mathtt{S}) \triangleq \mathtt{emp} \quad \text{otherwise.}\end{array}$$

Note that $\kappa - \kappa'$ removes a term κ' from κ, and $\mathtt{IsPtr}(\mathtt{v})$ determines if v is of type pointer.

Definition 3 (Well-formed formula) A separation formula is well-formed if:

- it is in a disjunctive normal form $\bigvee_{\mathtt{i}}(\exists\overline{\mathtt{v}}\cdot\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}})$;
- all occurrences of heap nodes are reachable from its accessible variables, S. That is, we have $\forall \mathtt{i}\cdot\kappa_{\mathtt{i}} = \mathtt{reach}(\kappa_{\mathtt{i}},\gamma_{\mathtt{i}},\mathtt{S})$ modulo associativity and commutativity of the separation conjunction $*$.

The primary significance of the well-formed condition is that all heap nodes of a heap constraint are reachable from accessible variables. This allows the entailment checking procedure to correctly match nodes from the consequent with nodes from the antecedent of an entailment relation.

Definition 4 (Well-founded Predicate) A shape predicate is said to be *well-founded* if it satisfies the following conditions:

- its body is a well-formed formula;
- given u as the special parameter which denotes the predicate's root pointer, if the body of the predicate contains some spatial formula, then u must be a heaplet of the form $\mathtt{u}{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}\rangle$.

Informally, the primary significance of the well-founded predicates is to ensure the monotonicity of the predicates which in turn guarantees the existence of a descending chain of unfoldings. The need for this well-founded conditions is clearer in Sec. 1.3 while showing how SLEEK solves the proof obligations generated by HIP.

1.2.3 Forward Verification Rules

The proof system is formalized using Hoare triples of the following form: $\vdash \{\Delta_{\mathtt{pre}}\}\ \mathtt{e}\ \{\Delta_{\mathtt{post}}\}$, where e is the expression to be verified, the pre-state $\Delta_{\mathtt{pre}}$ is given and the post-state $\Delta_{\mathtt{post}}$ is computed. We next generalize this triple to support a set of possible post-states: $\vdash \{\Delta_{\mathtt{pre}}\}\ \mathtt{e}\ \{S\}$, where S is a residual set of heap states discovered by the proof-search-based strategy adopted during the verification process. The verification is said to have succeeded with $\Delta_{\mathtt{pre}}$ as prestate if the residual set S is non-empty, and failed otherwise.

The verification rules are given in Fig. 1.4. The proof system engages a special variable res to denote the result of evaluating the target expression. ρ represents substitution, and $\mathcal{P}$ is the program being verified. The rules FIELD–READ, FIELD–UPDATE, METH–DEF, METH–CALL discharge proof obligations in the form of entailment relations written as $\Delta_1 \vdash \Delta_2 * S$ and read as "if Δ_1 is true, then Δ_2 is also true with the possible residual states S". In other words, the residual states are the witnesses that the respective proof obligation holds. The technicalities behind the entailment checks are postponed to Sec. 1.3.

Most of the verification rules are standard. METH–DEF shows how to verify a method definition annotated with p pairs of specifications. For each pair i of specifications, the verifier starts by assuming the method's i-th precondition, namely $\Phi^{\mathtt{i}}_{\mathtt{pr}}$. It then incrementally applies the other verification rules to verify that symbolically executing the method's body e leads to a set of post-states $S^{\mathtt{i}}_1$ each of which entails the corresponding post-condition $\Phi^{\mathtt{i}}_{\mathtt{po}}$. To note that a method takes in m pass-by-reference parameters and n − m pass-by-value parameters. To capture how the method's body changes the values of the pass-by-reference parameters, we use the primed notation, e.g. for a parameter v, we use $\mathtt{v}'$ to denote its latest (current) value. The *nochange* function initializes the current values of parameters to their initial (unprimed) values (the precondition $\Phi^{\mathtt{i}}_{\mathtt{pr}}$ is given only in terms of unprimed variables). At the end of the procedure, the current (primed) values of the pass-by-value parameters are existentially quantified from the post-state $\Phi^{\mathtt{i}}_{\mathtt{po}}$ so that their values are not visible by the postcondition, hence by the callers of the procedure.

METH–CALL is the rule for method call. At a high level, if a pre-condition, $\Phi^{\mathtt{i}}_{\mathtt{pr}}$, is satisfied at the call site then its corresponding post-condition, $\Phi^{\mathtt{i}}_{\mathtt{po}}$, is added to the state. The pass-by-value parameters, V, are equated to their initial values through the *nochange* function, since their final values are not visible to the method's callers. The residual heap state for each pair of specification, $S^{\mathtt{i}}$, from checking the method's i-th pre-condition

$$\frac{}{\vdash \{\Delta\}\ \mathtt{k}^{\tau}\ \{S = \{\Delta \wedge \mathtt{res}{=}\mathtt{k}^{\tau}\}\}}\ [\text{CONST}] \qquad \frac{}{\vdash \{\Delta\}\ \mathtt{v}\ \{S{=}\{\Delta \wedge \mathtt{res}{=}\mathtt{v}\}\}}\ [\text{VAR}] \qquad \frac{\vdash \{\Delta\}\ \mathtt{e}\ \{S\}}{\vdash \{\Delta\}\ \mathtt{t}\ \mathtt{v};\ \mathtt{e}\ \{\exists\, \mathtt{v}, \mathtt{v}' \cdot S\}}\ [\text{LOCAL}]$$

$$\frac{\vdash \{\Delta\}\ \mathtt{e}\ \{S_1\} \qquad S_2{=}\exists \mathtt{res}\cdot(S_1 \wedge \mathtt{v}'{=}\mathtt{res})}{\vdash \{\Delta\}\ \mathtt{v}{:=}\mathtt{e}\ \{S_2\}}\ [\text{ASSIGN}] \qquad \frac{\Delta \vdash \mathtt{v} \mapsto \mathtt{d}\langle \mathtt{v}_1, .., \mathtt{v}_n\rangle * S_1 \qquad S_1 \neq \emptyset \qquad \mathtt{fresh}\ \mathtt{v}_1..\mathtt{v}_n \qquad S_2 = \exists \mathtt{v}_1..\mathtt{v}_n \cdot (S_1 * \mathtt{v} \mapsto \mathtt{d}\langle \mathtt{v}_1, .., \mathtt{v}_n\rangle \wedge \mathtt{res}{=}\mathtt{v}_i)}{\vdash \{\Delta\}\ v.f_i\ \{S_2\}}\ [\text{FIELD-READ}]$$

$$\frac{S{=}\{\Delta * \mathtt{res} \mapsto \mathtt{d}\langle \mathtt{v}_1, .., \mathtt{v}_n\rangle\}}{\vdash \{\Delta\}\ \mathtt{new}\ \mathtt{d}(\mathtt{v}_1, .., \mathtt{v}_n)\ \{S\}}\ [\text{NEW}] \qquad \frac{\Delta \vdash \mathtt{v} \mapsto \mathtt{d}\langle \mathtt{v}_1, .., \mathtt{v}_n\rangle * S_1 \qquad S_1 \neq \emptyset \qquad \mathtt{fresh}\ \mathtt{v}_1..\mathtt{v}_n \qquad S_2 = \exists \mathtt{v}_1..\mathtt{v}_n \cdot (S_1 * \rho(\mathtt{v}' \mapsto \mathtt{d}\langle \mathtt{v}_1, .., \mathtt{v}_n\rangle)) \qquad \rho{=}[\mathtt{v}_0/\mathtt{v}_i]}{\vdash \{\Delta\}\ \mathtt{v}.\mathtt{f}_i{:=}\mathtt{v}_0\ \{S_2\}}\ [\text{FIELD-UPDATE}]$$

$$\frac{\vdash \{\Delta\}\ \mathtt{e}\ \{S_1\} \qquad S{=}\exists \mathtt{v} \cdot (\exists \mathtt{res} \cdot (S_1 \wedge \mathtt{v}{=}\mathtt{res}))}{\vdash \{\Delta\}\ \mathtt{return}\ \mathtt{e}\ \{S\}}\ [\text{RETURN}]$$

$$\frac{\vdash \{\Delta \wedge \mathtt{b}\}\ \mathtt{e}_1\ \{S_1\} \qquad \vdash \{\Delta \wedge \neg \mathtt{b}\}\ \mathtt{e}_2\ \{S_2\}}{\vdash \{\Delta\}\ \mathtt{if}\ (\mathtt{b})\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2\ \{S_1 \vee S_2\}}\ [\text{IF}] \qquad \frac{\vdash \{\Delta\}\ \mathtt{e}_1\ \{S_1\} \qquad \vdash \{S_1\}\ \mathtt{e}_2\ \{S_2\}}{\vdash \{\Delta\}\ \mathtt{e}_1; \mathtt{e}_2\ \{S_2\}}\ [\text{SEQ}]$$

$$\frac{\mathtt{V}{=}\{\mathtt{v}_m, .., \mathtt{v}_n\} \qquad \mathtt{W}{=}\{\mathtt{v}' \mid \mathtt{v} \in \mathtt{V}\} \qquad \forall \mathtt{i} = 1, .., \mathtt{p} \cdot (\vdash \{\Phi^i_{pr} \wedge \mathit{nochange}(\mathtt{V})\}\ \mathtt{e}\ \{S^i_1\} \quad \exists \mathtt{W} \cdot S^i_1 \vdash \Phi^i_{po} * S^i_2 \quad S^i_2 \neq \emptyset)}{\vdash \mathtt{t}\ mn((\mathtt{ref}\ \mathtt{t}_j\ \mathtt{v}_j)^{m-1}_{j=1}, (\mathtt{t}_j\ \mathtt{v}_j)^{n}_{j=m})\{\mathtt{requires}\ \Phi^i_{pr}\ \ \mathtt{ensures}\ \Phi^i_{po}\}^{p}_{i=1}\{\mathtt{e}\}}\ [\text{METH-DEF}]$$

$$\frac{\mathtt{t}\ mn((\mathtt{ref}\ \mathtt{t}_j\ \mathtt{v}^0_j)^{m-1}_{j=1}, (\mathtt{t}_j\ \mathtt{v}^0_j)^{n}_{j=m})\ \{\mathtt{requires}\ \Phi^i_{pr}\ \ \mathtt{ensures}\ \Phi^i_{po}\}^{p}_{i=1}\ \{\mathtt{e}\} \in \mathcal{P} \qquad \rho{=}[\mathtt{v}_j/\mathtt{v}^0_j]^{n}_{j=1} \qquad \forall \mathtt{i}{=}1, .., \mathtt{p} \cdot (\Delta \vdash \rho\Phi^i_{pr} * S_i) \qquad S = \bigcup^{p}_{i=1}(S_i * \rho\Phi^i_{po}) \qquad S \neq \emptyset}{\vdash \{\Delta\}\ mn(\mathtt{v}_1, .., \mathtt{v}_n)\ \{S\}}\ [\text{METH-CALL}]$$

Fig. 1.4 Forward Verification Rules

is composed with its corresponding post-condition to become the post-state, S , of the method call.

For brevity, we lifted the binary operations normally used for formulae composition, namely $*$, $\wedge$, $\vee$, to composition of sets and formulae in order to precisely capture the residue of each proof rule (note that the separation conjunction operator $*$ is commutative, associative, and distributive over disjunction). The normalization patterns that HIP/SLEEK often engages are presented in Fig. 1.5.

$$
\begin{array}{llll}
\pi \wedge S & \equiv \{\Delta_s \wedge \pi \mid \Delta_s \in S\} & (\kappa_1 \wedge \pi_1) * (\kappa_2 \wedge \pi_2) & \equiv \kappa_1 * \kappa_2 \wedge \pi_1 \wedge \pi_2 \\
S \wedge \pi & \equiv \{\Delta_s \wedge \pi \mid \Delta_s \in S\} & S_1 \vee S_2 & \equiv \{\Delta_1 \vee \Delta_2 \mid \Delta_1 \in S_1,\ \Delta_2 \in S_2\} \\
\Delta * S & \equiv \{\Delta_s * \Delta \mid \Delta_s \in S\} & \exists v \cdot (S \wedge \pi) & \equiv \{\exists v \cdot (\Delta_s \wedge \pi) \mid \Delta_s \in S\} \\
S * \Delta & \equiv \{\Delta_s * \Delta \mid \Delta_s \in S\} & (\exists x \cdot \Delta) \wedge \pi & \equiv \exists y \cdot ([y/x]\Delta) \wedge \pi \\
(\Delta_1 \vee \Delta_2) \wedge \pi & \equiv (\Delta_1 \wedge \pi) \vee (\Delta_2 \wedge \pi) & (\exists x \cdot \Delta_1) * \Delta_2 & \equiv \exists y \cdot ([y/x]\Delta_1) * \Delta_2
\end{array}
$$

Fig. 1.5 Normalization Rules

1.3 SLEEK – Entailment Checking

The proof obligations generated in Sec 1.2, abbreviated as *heap entailments*, are handled by our entailment prover SLEEK. The formulas of the entailments are a combination of separation logic and heap-independent logics, and are of the form

$$\Delta_A \vdash^{\kappa}_{V} \Delta_C * \Delta_R$$

which is a consequence (or shortcut) for

$$\kappa * \Delta_A \vdash \exists V \cdot (\kappa * \Delta_C) * \Delta_R$$

To prove the above entailment, we need to check whether the heaplet modelled by the antecedent Δ_A is sufficiently precise to cover all heaplets modelled by the consequent Δ_C. Furthermore, we also aim to compute the *residual heap state* Δ_R (a.k.a the "frame" condition [9]), which represents what was not consumed from the antecedent after matching up with the formula form the consequent. κ is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified variables from the consequent. Note that κ and V are derived during the entailment proof. The entailment checking procedure is initially invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$. The entailment proving rules are explained as follows.

Handling entailments of disjunctive formulas

Firstly, we present the reduction from entailment between disjunctive formulas with existential quantifiers to entailment between quantifier-free conjunctive formulas.

Removing disjunction. An entailment with a disjunctive antecedent succeeds if both disjuncts entail the consequent (ENT–LHS–OR). Conversely, an entailment with disjunctive consequent succeeds if either one of the disjuncts succeeds (ENT–RHS–OR).

$$\frac{[\underline{\text{ENT–LHS–OR}}]}{\Delta_1 \vdash^{\kappa}_{V} \Delta_3 * \Delta_4 \qquad \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * \Delta_5}{\Delta_1 \vee \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * (\Delta_4 \vee \Delta_5)} \qquad \frac{[\underline{\text{ENT–RHS–OR}}]}{\Delta_1 \vdash^{\kappa}_{V} \Delta_i * \Delta_i^{R} \qquad i \in \{2,3\}}{\Delta_1 \vdash^{\kappa}_{V} (\Delta_2 \vee \Delta_3) * \Delta_i^{R}}$$

Removing existential quantifiers. Existentially quantified variables from the antecedent are simply lifted out of the entailment relation by replacing them with fresh variables (ENT–LHS–EX). On the other hand, we keep track of the existential variables coming from the consequent by adding them to V (ENT–RHS–EX).

$$\frac{[\text{ENT-RHS-EX}]}{\Delta_1 \vdash^{\kappa}_{V\cup\{\mathtt{w}\}} ([\mathtt{w}/\mathtt{v}]\Delta_2) * \Delta \quad \mathtt{fresh\ w}}{\Delta_1 \vdash^{\kappa}_{V} (\exists \mathtt{v} \cdot \Delta_2) * \Delta} \qquad \frac{[\text{ENT-LHS-EX}]}{[\mathtt{w}/\mathtt{v}]\Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta \quad \mathtt{fresh\ w}}{\exists \mathtt{v} \cdot \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta}$$

Handling entailments of conjunctive formulas

We now present the reduction of entailment between two quantifier-free conjunctive formulae to entailment between two pure formulae.

Handling consequent with empty heap. The base case for our entailment checker occurs when the consequent is a pure formula, in which case the ENT-EMP rule is applied. The rule first approximates the antecedent of the entailment, including the *footprint* heap formulae κ that have been matched previously. The approximation is done by the XPure function which taken a heap formula as input and outputs its pure approximation, e.g. the information that a pointer p points to a valid memory location, $\mathtt{p}{\mapsto}\mathtt{node}\langle \mathtt{x}, \mathtt{y}\rangle$, is approximated by the pure formula $\mathtt{p} \neq \mathtt{null}$. The full definition of XPure is deferred to the end of this sub-chapter. The checker next invokes an off-the-shelf theorem prover to check if the approximation of the antecedent implies the heap-independent consequent.

$$\frac{[\text{ENT-EMP}]}{\mathtt{XPure}_n(\kappa_1 * \kappa) \wedge \pi_1 \Rightarrow \exists V \cdot \pi_2}{\kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} \mathtt{emp} \wedge \pi_2 * (\kappa_1 \wedge \pi_1)}$$

Matching and removing heap nodes from the antecedent and the consequent. The rule ENT-MATCH successively matches up heap nodes that can be proven aliased.

$$\frac{\begin{array}{c}[\text{ENT-MATCH}]\\ \mathtt{XPure}_n(\mathtt{u}_1{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}_1\rangle * \kappa_1 \wedge \pi_1) \Rightarrow \mathtt{u}_1{=}\mathtt{u}_2\\ \rho = [\overline{\mathtt{v}}_1/\overline{\mathtt{v}}_2] \qquad \kappa_1 \wedge \pi_1 \wedge \mathtt{freeEqn}(\rho, V) \vdash^{\kappa * \mathtt{u}_1 \mapsto \mathtt{d}\langle\overline{\mathtt{v}}_1\rangle}_{V-\{\overline{\mathtt{v}}_2\}} \rho(\kappa_2 \wedge \pi_2) * \Delta\end{array}}{\mathtt{u}_1{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}_1\rangle * \kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} (\mathtt{u}_2{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}_2\rangle * \kappa_2 \wedge \pi_2) * \Delta}$$

In the above rule, the condition $\mathtt{XPure}_n(\mathtt{u}_1{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}_1\rangle * \kappa_1 \wedge \pi_1) \Rightarrow \mathtt{u}_1{=}\mathtt{u}_2$ checks whether $\mathtt{u}_1$ and $\mathtt{u}_2$ are aliasing based on the information in the antecedent. If two atomic heap formulas have the same name, that is, they are either two heaplets of the same type, or two instances of the same predicate, we require their fields or, respectively, arguments to be the same. The unification of the two heap formulas is accomplished by the application of substitution ρ to the remaining of the consequent. Since $\overline{\mathtt{v}}_2$ is substituted away, it is removed from the list of existentially quantified variables.

When a match occurs and an argument of the heap node coming from the consequent is free, the entailment procedure binds the argument to the corresponding variable from the antecedent and moves the equality to the antecedent. In our system, free variables in consequent are variables from method preconditions. These bindings play the role of parameter instantiations during forward reasoning and can be accumulated into the antecedent to allow the subsequent program state (from the residual heap state) to be aware of their instantiated values. This process is formalized by function freeEqn, where V is the set of existentially quantified variables:

$$\begin{array}{l}\mathtt{freeEqn}([\mathtt{u}_i/\mathtt{v}_i]_{i=1}^n, V) \triangleq \mathtt{let}\ \pi_i = (\mathtt{if}\ \mathtt{u}_i \in V\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ \mathtt{v}_i = \mathtt{u}_i)\\ \qquad\qquad\qquad\qquad \mathtt{in} \bigvee_{i=1}^n \pi_i\end{array}$$

For soundness, we perform a preprocessing step to ensure that variables appearing as arguments of heap nodes and predicates are (i) distinct and (ii) if they are free, they do not appear in the antecedent by adding (existentially quantified) fresh variables and equalities. This guarantees that the formula generated by `freeEqn` does not introduce any additional constraints over existing variables in the antecedent.

Unfolding a shape predicate in the antecedent. If a predicate instance in the antecedent is aliased with a point-to predicate in the consequent, we unfold it by using the rule ENT–UNFOLD below.

$$\frac{[\underline{\text{ENT–UNFOLD}}]\quad \texttt{XPure}_n(\texttt{P}(\texttt{u}_1,\overline{\texttt{v}}_1)*\kappa_1\wedge\pi_1)\Rightarrow \texttt{u}_1{=}\texttt{u}_2 \quad \texttt{unfold}(\texttt{P}(\texttt{u}_1,\overline{\texttt{v}}_1))*\kappa_1\wedge\pi_1\vdash^{\kappa}_{V}(\texttt{u}_2{\mapsto}\texttt{d}\langle\overline{\texttt{v}}_2\rangle*\kappa_2\wedge\pi_2)*\Delta}{\texttt{P}(\texttt{u}_1,\overline{\texttt{v}}_1)*\kappa_1\wedge\pi_1\vdash^{\kappa}_{V}(\texttt{u}_2{\mapsto}\texttt{d}\langle\overline{\texttt{v}}_2\rangle*\kappa_2\wedge\pi_2)*\Delta}$$

where the function `unfold` is defined as follows:

$$\frac{\texttt{P}(\overline{\texttt{v}})\ \triangleq\ \Phi\ \text{inv}\ \pi}{\texttt{unfold}(\texttt{P}(\overline{\texttt{u}}))\ \triangleq\ [\overline{\texttt{u}}/\overline{\texttt{v}}]\Phi}$$

The above rule basically replaces the predicate instance by its predicate definition, normalizes the resulting formula, and resumes entailment checking. Each unfolding either exposes an object that matches the object in the consequent, or reduces the atomic heap formula in the antecedent $\texttt{u}_1{\mapsto}\texttt{d}\langle\overline{\texttt{v}}_1\rangle$ to a pure formula. The former case results in a reduction of the consequent by using ENT–MATCH. In the latter case, the entailment either (i) fails immediately since the checker is unable to find an aliased heap node, or (ii) if the obtained pure formula reveals additional aliasing information, the entailment checker continues with a new aliased heap node from the antecedent. If the new aliased heap node is an object, a match and thus a reduction of the consequent occurs. Otherwise, a new unfolding is triggered. This process cannot go forever as every time it happens, one predicate from the antecedent is removed and no new predicate instance is generated.

Folding against a shape predicate in the consequent. If a predicate instance in the consequent does not have a matching predicate instance in the antecedent, we attempt to generate one by folding the antecedent (ENT–FOLD).

$$\frac{[\underline{\text{ENT–FOLD}}]\quad (\Delta^r,\kappa^r,\pi^r)\in\texttt{fold}^{\kappa}(\texttt{u}_1{\mapsto}\texttt{d}\langle\overline{\texttt{v}}_1\rangle*\kappa_1\wedge\pi_1,\texttt{P}(\texttt{u}_2,\overline{\texttt{v}}_2)) \quad (\pi^1,\pi^c)=\texttt{split}_V^{\{\overline{\texttt{v}}_2\}}(\pi^r) \quad \texttt{XPure}_n(\texttt{u}_1{\mapsto}\texttt{d}\langle\overline{\texttt{v}}_1\rangle*\kappa_1*\pi_1\Rightarrow\texttt{u}_1{=}\texttt{u}_2) \quad \Delta^r\wedge\pi^a\vdash^{\kappa^r}_{V}(\kappa_2\wedge\pi_2\wedge\pi^c)*\Delta}{\texttt{u}_1{\mapsto}\texttt{d}\langle\overline{\texttt{v}}_1\rangle*\kappa_1\wedge\pi_1\vdash^{\kappa}_{V}(\texttt{P}(\texttt{u}_2,\overline{\texttt{v}}_2)*\kappa_2\wedge\pi_2)*\Delta}$$

where the function `fold` is defined as follows:

$$\frac{\texttt{P}(\overline{\texttt{v}})\ \triangleq\ \Phi\ \text{inv}\ \pi \quad \kappa\wedge\pi\vdash^{\kappa'}_{\{\texttt{u},\overline{\texttt{v}}\}}[\overline{\texttt{u}}/\overline{\texttt{v}}]\Phi*\{(\Delta_i,\kappa_i,V_i,\pi_i)\}_{i=1}^{n} \quad W_i=V_i-\{\overline{\texttt{v}},\texttt{u}\}}{\texttt{fold}^{\kappa}(\kappa\wedge\pi,\texttt{P}(\overline{\texttt{u}}))\ \triangleq\ \{(\Delta_i,\kappa_i,\exists W_i\cdot\pi_i)\}_{i=1}^{n}}$$

Some heap nodes from κ are removed by the entailment procedure in the *fold* definition so as to match with the heap formula of the predicate $\texttt{P}(\overline{\texttt{v}})$. This requires a special version of entailment that returns three extra things: (i) the consumed heap nodes, κ_i, (ii) the existential variables used, W_i, and (iii) the final consequent, Δ_i. The final consequent is used to return a constraint for $\{\overline{v}\}$ via $\exists W_i\cdot\pi_i$. A set of n answers is returned by the fold step since we allow it to explore multiple ways of matching up with its disjunctive

heap state (as depicted by the entailment's special set residue $\{(\Delta_i, \kappa_i, V_i, \pi_i)\}_{i=1}^{n}$ – note that only Δ_i is an actual heap residue, the rest of the tuple's elements are additional information required to fill up by the ENT–FOLD entailment rule).

When a fold against a predicate $\mathtt{p}(\mathtt{u}_2, \overline{\mathtt{v}}_2)$ is performed, the constraints related to variables $\overline{\mathtt{v}}_2$ are significant. The `split` function projects these constraints out and differentiates those constraints based on free variables, distinguishing what can be assumed, π_i^a, from what needs to be proved to hold true, (π_i^c). The formal definition of `split` is as follows:

$$\begin{array}{l}\mathtt{split}_V^{\{\overline{\mathtt{v}}_2\}}(\bigwedge_{i=1}^{n}\pi_i^r) \triangleq \mathtt{let}\ \pi_i^a, \pi_i^c = \mathtt{if}\ FV(\pi_i^r) \cap \overline{\mathtt{v}}_2 = \emptyset\ \mathtt{then}\ (\mathtt{true}, \mathtt{true}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{else\ if}\ FV(\pi_i^r) \cap V = \emptyset\ \mathtt{then}\ (\pi_i^r, \mathtt{true}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{else}\ (\mathtt{true}, \pi_i^r) \\ \qquad\qquad\qquad\qquad\qquad \mathtt{in}\ (\bigwedge_{i=1}^{n}\pi_i^a, \bigwedge_{i=1}^{n}\pi_i^c)\end{array}$$

In other words, `split` helps distinguishing between the pure constraints which are the result of the fold operation introducing bindings for the parameters of the folded predicate, from the pure constraints which are in the definition of the folded predicate. The former are transferred to the antecedent, while the latter remain in the consequent since they need to be proven to hold.

Approximating a separation formula by a pure formula

In our entailment proof, the entailment between separation formulae is finally reduced to entailment between pure formulae by successively removing heap nodes from the consequent until only a pure formula remains. When this happens, the heap formula in the antecedent can be soundly approximated by the $\mathtt{XPure}_n$ function to obtain a pure (heap-independent) formula which can be discharged by a theorem prover. In this function, the index `n` indicates how precise the caller wants the approximation to be. The function $\mathtt{XPure}_n$ is recursively defined as follows:

- $\mathtt{XPure}_n(\mathtt{emp}) \triangleq \mathtt{true}$
- $\mathtt{XPure}_n(\mathtt{u}{\mapsto}\mathtt{d}\langle\overline{\mathtt{v}}\rangle) \triangleq \mathtt{u} \neq \mathtt{null}$
- $\mathtt{XPure}_n(\mathtt{P}(\overline{\mathtt{v}})) \triangleq \mathtt{Inv}_n(\mathtt{P}(\overline{\mathtt{v}}))$
- $\mathtt{XPure}_n(\kappa_1 * \kappa_2) \triangleq \mathtt{XPure}_n(\kappa_1) \wedge \mathtt{XPure}_n(\kappa_2)$
- $\mathtt{XPure}_n(\bigvee_{i=1}^{n}(\exists\overline{\mathtt{v}}_\mathtt{i} \cdot (\kappa_i \wedge \pi_i))) \triangleq \bigvee_{i=1}^{n}(\exists\overline{\mathtt{v}}_\mathtt{i} \cdot (\mathtt{XPure}_n(\kappa_i) \wedge \pi_i))$

In the above definition, $\mathtt{XPure}_n$ calls a related function $\mathtt{Inv}_n$ to compute the pure approximation. This function $\mathtt{Inv}_n$ is defined as follows.

$$\frac{\mathtt{P}(\overline{\mathtt{v}}) \triangleq \Phi\ \mathtt{inv}\ \pi_0}{\mathtt{Inv}_0(\mathtt{P}(\overline{\mathtt{v}})) \triangleq \pi_0} \qquad\qquad \frac{\mathtt{P}(\overline{\mathtt{v}}) \triangleq \Phi\ \mathtt{inv}\ \pi_0}{\mathtt{Inv}_n(\mathtt{P}(\overline{\mathtt{v}})) \triangleq \mathtt{XPure}_{n-1}(\Phi)}$$

Specifically, when `n`=0, $\mathtt{Inv}_n$ returns the user-supplied invariant. In the recursive case, `n`>0, $\mathtt{Inv}_n$ invokes $\mathtt{XPure}_{n-1}$ to compute a more precise invariant based on the body of the predicate. When the invariant is not provided, our system relies on those satisfiability solvers that can handle separation logic with inductive predicates [52, 55, 57].

```
1  data node {node prev; node next; }

3  void sll2dll(node x, node q)
4  { // S1 : H(x,q#)
5     if(x!=NULL) {
6           // S2 : x↦node⟨xp,xn⟩*Hp(xp,q#)*Hn(xn,q#)∧x≠null
7           x.prev=q;
8           // S3 : x↦node⟨q,xn⟩*Hp(xp,q#)*Hn(xn,q#)∧x≠null
9           sll2dll(x.next,x);
10          //S4 : x↦node⟨q,xn⟩*Hp(xp,q#)*G(xn,x#)∧x≠null
11    }
12   //S5 : H(x,q#)∧x=null
13 }
```

Fig. 1.6 Example

1.4 Specification Inference

Specification inference is a technique that uses static analysis to synthesize formal specifications in order to capture some properties of a given program. HIP infers specification to capture the shape of the manipulating heap regions (to ensure memory safety) as well as functional properties (e.g., constraints over pure variables) and non-functional properties (e..g, termination and non-termination) for heap-based programs. The specification inference mechanism in HIP/SLEEK system is based on a new second-order bi-abductive entailment with unknown predicates (a.k.a., second-order variables). It constructs sound interpretations for the unknown predicates during code verification [53, 91] by generating *relational assumptions* that capture predefined analysis properties for the unknown predicates. The specification is inferred incrementally: first with shape property and then other pure (non-shape) properties. Typically, shape-based specification is inferred through predicate definition derivation and normalization steps. Pure-based specification is inferred through pure-property extension and fixed point computation.

In the following, we first show our approach through an example (Subsection 1.4.1). After that, we describe the second-order formalism (Subsection 1.4.2). Finally, we present the Hoare logic for specification inference (Subsection 1.4.3).

1.4.1 Illustrative Example

As a running example, consider the method shown in Figure 1.6 which traverses a singly-linked list, and gradually changes the input list into a doubly-linked list.

To capture the specification that ensures the memory safety for this method, we could use the following inductive predicates.

$$\texttt{sllN(hd,n)} \equiv \texttt{emp}\wedge\texttt{hd=null} \wedge n{=}0 \vee \exists nx.\ \texttt{hd}\mapsto\texttt{node}\langle\_,\texttt{nx}\rangle{*}\texttt{sllN(nx,n}{-}1\texttt{)}$$
$$\texttt{dllN(hd,p,n)} \equiv \texttt{emp}\wedge\texttt{hd=null} \wedge n{=}0 \vee \exists nx.\ \texttt{hd}\mapsto\texttt{node}\langle\texttt{p},\texttt{nx}\rangle{*}\texttt{dllN(nx,hd,n}{-}1\texttt{)}$$

Here, the sllN predicate describes the shape of an acyclic singly-linked list pointed by hd. In its definition, the first disjunct corresponds to the case of an empty list, whereas the second one separates a list into two parts: the head $\texttt{hd}\mapsto\texttt{node}\langle\_,\texttt{nx}\rangle$, and the tail $\texttt{sllN}(\texttt{nx},\texttt{n}-1)$. Similarly, the dllN predicate describes the shape of an acyclic doubly-linked list pointed by hd. With these predicates, we can write a specification that ensures memory safety for this method through the following pair of pre/post conditions.

$$\texttt{requires sllN}(\texttt{x},\texttt{n}) \qquad \texttt{ensures dllN}(\texttt{x},\texttt{q},\texttt{n}) \wedge n{\geq}0$$

Once predicate definitions and pre/post specifications for the method are given, the automated verification system (as presented in the preceding sections) could verify that all memory accesses are safe, and that the post-condition of the method is ensured, namely that a doubly-linked list of the same size as the list in the pre-condition has been created. Our shape inference framework endeavours to undertake the reverse scenario where predicate definitions are not given a priori. Instead, to trigger the incremental inference in the HIP system, the end user only needs to annotate the following command:

$$\begin{array}{l}\texttt{infer}\ \ [@shape, @size, @pre, @post]\\ \texttt{requires}\ \ \texttt{true}\qquad \texttt{ensures}\ \ \texttt{true}\end{array}$$

where the parameter of the infer command describes all stages which are to be considered during the inference: $@shape$ for shape inference, $@size$ for the size property extension, and $@pre$ (resp., $@post$) for the pure constraint in the pre-condition (resp., post-condition) inference. In the following, we illustrate how the HIP system could infer the above specification via second-order bi-abduction [53, 91].

Shape Inference. The shape inference in HIP relies on capabilities of shape analysis. Given a program, the shape analysis infers shapes of dynamic linked data structures pointer by pointer at program locations that are required for memory safety. Shape analysis mechanisms typically infer specifications for memory safety with a predetermined set of shape predicates [6, 9, 10, 64]. Discovering arbitrary shape abstractions is challenging, as linked data structures span a wide variety of forms, from singly-linked lists, doubly-linked lists, circular lists, to tree-like data structures. Furthermore, such abstractions would also need to cater to various specializations, such as strictly non-empty structures or segmented structures (e.g. list/tree segments) with outward pointing references. We now show how to infer complicated shape specifications, from scratch, directly from heap-based programs.

By command $\texttt{infer}\ [@shape]$, HIP internally introduces two unknown predicates, H and G, as the place-holders to capture the shape specification for the pre-condition and the post-condition, respectively, in the following specification.

$$\begin{array}{l}\texttt{infer}\ \ [\texttt{H},\texttt{G}]\\ \texttt{requires}\ \ \texttt{H}(\texttt{x},\texttt{q}\#)\qquad \texttt{ensures}\ \ \texttt{G}(\texttt{x},\texttt{q}\#)\end{array}$$

Intuitively, it is meant to incorporate the inference capability (via infer) into a pair of pre/post-conditions (via requires/ensures). Here the inference will be applied to second-order variables $\texttt{H},\texttt{G}$. For clarity, we use $\texttt{H}_1,\texttt{H}_2,\ldots$ as names for unknown pre-predicates and $\texttt{G}_1,\texttt{G}_2,\ldots$ as names for unknown post-predicates. Note that we control the instantiation of variables by using annotation # to mark those variables that are

disallowed from being instantiated. This scheme ensures that each pointer is instantiated at most once, thus avoiding `false` from being inferred.

The three steps of our shape inference are: (i) Derive relational assumptions during code verification using second-order bi-abduction; (ii) Derive predicate definitions from relational assumptions by abductive or equivalence-preserving transformation; and (iii) Normalize and, whenever possible, substitute available predicates for unknowns to support reuse. Firstly, we apply code verification using these pre/post specification and attempt to infer program states S_1, .., S_5 (of the form of separation logic formulas as shown in Figure 1.6) as well as to collect a set of verification conditions (VCs) that must hold to ensure memory-safety. These conditions also ensure that the pre-conditions of each method call are satisfied, and that the post-condition is ensured at the end of its method body. Each VC is a relational assumption of the form $\Delta_a \mid \Delta_g \Rightarrow \Delta_c$ where Δ_a, Δ_g, and Δ_c may contain the unknown predicates.

For `sll2dll` method, HIP can derive the following four relational assumptions:

$$\begin{array}{ll}
\mathtt{H(x,q\#)} \wedge \mathtt{x{\neq}null} & \Rightarrow \mathtt{x{\mapsto}node\langle x_p, x_n\rangle{*}H_p(x_p,q\#){*}H_n(x_n,q\#)} \\
\mathtt{H_n(x_n,q\#)} \mid \mathtt{x{\mapsto}node\langle q, x_n\rangle} & \Rightarrow \mathtt{H(x_n,x\#)} \\
\mathtt{H(x,q\#)} \wedge \mathtt{x{=}null} & \Rightarrow \mathtt{G(x,q\#)} \\
\mathtt{x{\mapsto}node\langle q,x_n\rangle{*}G(x_n,x\#)} & \Rightarrow \mathtt{G(x,q\#)}
\end{array}$$

The first assumption is generated by the access to the `x.prev` field, which mandated that $\mathtt{H(x,q\#)}$ generates a node under a condition $\mathtt{x{\neq}null}$, from the `then` branch of its conditional statement. The second assumption is for modularly proving the precondition of the recursive call to $\mathtt{sll2dll(x.next,x)}$. The last two assumptions are required to ensure the ascribed post-condition of the `sll2dll` method hold for both branches (the else branch is implicit).

There are several new features that we have designed into our relational assumptions. Firstly, we may introduce new unknown predicates, such as $\mathtt{H_p}$ and $\mathtt{H_n}$, so as to provide possible expansion (or instantiation) points for yet to be explored data fields. Secondly, we use a #-annotation scheme to carefully ensure that each pointer is instantiated at most once. This is to prevent us from accidentally creating an unsatisfiable `false` state when a pointer is instantiated more than once. Our annotation scheme is fully automatic as we have developed a simple static analysis to determine the #-annotation for the initial pre/post predicates. In the case of the `sll2dll` example, we will mark the second parameter of both predicates `H` and `G` as non-instantiating, i.e. $\mathtt{H(x,q\#)}$ and $\mathtt{G(x,q\#)}$, since it is never field-accessed in the code. Thirdly, we allow a special heap guard introduced after $\mid$ (e.g. $\mathtt{x{\mapsto}node\langle q, x_n\rangle}$ in the second assumption), to clearly describe the heap context where the relational assumption was applicable. This heap guard is used to guide the instantiation of some parameters (e.g. `x#`), and is critical for properly handling back-pointers (such as previous or parent pointers). The guard simply qualifies $\mathtt{H_n(x_n,q\#){\Rightarrow}H(x_n,x\#)}$, requiring the availability of heap context $\mathtt{x{\mapsto}node\langle q, x_n\rangle}$.

Based on this set of relational assumptions, it is possible to construct an interpretation for the unknown predicates `H`, `G`, and $\mathtt{H_n}$ that ensure the validity of these assumptions. As for $\mathtt{H_p}$, we would refrain from imposing any interpretation since its contents are not being accessed by the `sll2dll` method. We refer to predicates without any interpretation

as *dangling predicates* since they denote dangling references that are not being accessed by the current method.

Our next phase uses a predicate derivation procedure to transform (by either equivalence-preserving or abductive steps) each set of relational assumptions into its corresponding set of predicate definitions. This procedure uses sound heuristics, and thus may occasionally fail to discover best possible predicate definitions. For our running example, we can generate the following three predicate definitions, after replacing each dangling predicate, e.g. $\mathtt{H_p}(\mathtt{x_p},\mathtt{q})$, that came from expanding a field by a unique global variable (e.g. $\mathcal{D}_\mathtt{p}$) which denotes the set of (unconstrained) references from its (e.g. `prev`) field. For this example, the dangling reference is captured recursively within $\mathtt{H}(\mathtt{x},\mathtt{q})$, and denotes multiple `prev` fields of the singly-linked list that were overwritten in the post-condition. Note that both $\mathtt{H}(\mathtt{x},\mathtt{q}\#) \wedge \mathtt{x{=}null} \Rightarrow \mathtt{emp}$ and $\mathtt{emp} \wedge \mathtt{x{=}null} \Rightarrow \mathtt{G}(\mathtt{x},\mathtt{q}\#)$ are derived by splitting from a single relational assumption $\mathtt{H}(\mathtt{x},\mathtt{q}\#) \wedge \mathtt{x{=}null} \Rightarrow \mathtt{G}(\mathtt{x},\mathtt{q}\#)$.

The definition of $\mathtt{H_n}$ is in a special guarded form to facilitate the instantiation of back-pointers. Instances of such guarded definitions can always be inlined, so that they are never required in our final specifications. (In both assumption $\Delta_\mathtt{lhs} \Rightarrow \Delta_\mathtt{rhs}$ and definition $\Delta_\mathtt{lhs} \equiv \Delta_\mathtt{rhs}$, the free variables from $FV(\Delta_\mathtt{rhs}) - FV(\Delta_\mathtt{lhs})$ are implicitly existentially quantified.)

$$\begin{array}{l}
\mathtt{H}(\mathtt{x},\mathtt{q}) \;\equiv\; \mathtt{emp} \wedge \mathtt{x{=}null} \vee \mathtt{x} \mapsto \mathtt{node}\langle \mathcal{D}_\mathtt{p}, \mathtt{x_n}\rangle * \mathtt{H}(\mathtt{x_n},\mathtt{x}) \\
\mathtt{H_n}(\mathtt{x_n},\mathtt{q}) \mid \mathtt{x} \mapsto \mathtt{node}\langle \mathtt{q}, \mathtt{x_n}\rangle \;\equiv\; \mathtt{H}(\mathtt{x_n},\mathtt{x}) \\
\mathtt{G}(\mathtt{x},\mathtt{q}) \;\equiv\; \mathtt{emp} \wedge \mathtt{x{=}null} \vee \mathtt{x} \mapsto \mathtt{node}\langle \mathtt{q}, \mathtt{x_n}\rangle * \mathtt{G}(\mathtt{x_n},\mathtt{x})
\end{array}$$

After deriving all predicate definitions, we proceed with the last phase on normalization to simplify and reuse predicates, where possible [53, 56]. This phase would eliminate the second parameter of H and the unused predicate, $\mathtt{H_n}(\mathtt{x},\mathtt{p})$, yielding:

$$\begin{array}{l}
\mathtt{H}(\mathtt{x},\mathtt{q}) \equiv \mathtt{H_2}(\mathtt{x}) \\
\mathtt{H_2}(\mathtt{x}) \;\equiv\; \mathtt{emp} \wedge \mathtt{x{=}null} \vee \mathtt{x} \mapsto \mathtt{node}\langle \mathcal{D}_\mathtt{p}, \mathtt{x_n}\rangle * \mathtt{H_2}(\mathtt{x_n}) \\
\mathtt{G}(\mathtt{x},\mathtt{q}) \equiv \mathtt{emp} \wedge \mathtt{x{=}null} \vee \mathtt{x} \mapsto \mathtt{node}\langle \mathtt{q}, \mathtt{x_n}\rangle * \mathtt{G}(\mathtt{x_n},\mathtt{x})
\end{array}$$

Note that $\mathtt{H_2}(\mathtt{x})$ is a specialized version of $\mathtt{H}(\mathtt{x},\mathtt{q})$ without the redundant `q` parameter. Whenever possible, we would also attempt to reuse existing predicate definitions to support shorter specifications and improve analysis/verification. If the following predicates `sll` and `dll` definitions have been supplied earlier,

$$\begin{array}{l}
\mathtt{sll}(\mathtt{hd}) \;\equiv \mathtt{emp} \wedge \mathtt{hd{=}null} \vee \exists nx.\, \mathtt{hd} \mapsto \mathtt{node}\langle _, \mathtt{nx}\rangle * \mathtt{sll}(\mathtt{nx}) \\
\mathtt{dll}(\mathtt{hd},\mathtt{p}) \equiv \mathtt{emp} \wedge \mathtt{hd{=}null} \vee \exists nx.\, \mathtt{hd} \mapsto \mathtt{node}\langle \mathtt{p}, \mathtt{nx}\rangle * \mathtt{dll}(\mathtt{nx},\mathtt{hd})
\end{array}$$

we would relate our new definitions to these prior definitions, as follows.

$$\mathtt{H_2}(\mathtt{x}) \equiv \mathtt{sll}(\mathtt{x}) \qquad\qquad \mathtt{G}(\mathtt{x},\mathtt{q}) \equiv \mathtt{dll}(\mathtt{x},\mathtt{q})$$

Pure-Property Extension. By the `infer` [@*size*] command, the user triggers an inference for the size constraints. Through the predicate extension mechanism, HIP injects the size property (captured by `n`) into the predicate `sll` to derive a predicate `llN` as:

$$\begin{array}{l}\texttt{pred llN(hd,n)} \triangleq (\texttt{emp}\wedge\texttt{hd=null} \wedge \texttt{n=0})\\ \vee \exists\, \texttt{nx,m}\cdot(\texttt{hd}\mapsto\texttt{node}\langle \_ , \texttt{nx}\rangle * \texttt{llN}\langle \texttt{nx,m}\rangle \wedge \texttt{n=m+1}).\end{array}$$

Similarly, the `dllN` predicate is generated as follows:

$$\begin{array}{l}\texttt{pred dllN(hd,p,n)} \triangleq (\texttt{emp}\wedge\texttt{hd=null} \wedge \texttt{n=0})\\ \vee \exists\, \texttt{nx,m}\cdot(\texttt{hd}\mapsto\texttt{node}\langle \texttt{q,nx}\rangle * \texttt{dllN}\langle \texttt{nx,hd,m}\rangle \wedge \texttt{n=m+1}).\end{array}$$

The current size extension mechanism simply adds a depth computation for each of the recursive predicates in separation logic.

Pure Inference. By the command `infer` $[@pre, @post]$, (i.e. the user would like to infer the specification with the size of each list), HIP internally strengthens the pre/post specification for `sll2dll` to include uninterpreted relations: `P(a)` in the precondition and `Q(a,b)` in the postcondition as follows.

```
infer [P,Q]
requires sllN(x,q,a) ∧ P(a)
ensures  dllN(x,q,b) ∧ Q(a,b);
```

Here, the inference will be applied to the second-order variables `P,Q`. They are meant to capture the relationship between the newly-introduced variables `a,b` denoting size properties of linked lists. Uninterpreted relations in the precondition should be as weak as possible, while ones in the postcondition should be as strong as possible. By analysing the `sll2dll` code, HIP gathers the following relational assumptions:

$$\begin{array}{l}\texttt{P(a)} \wedge \texttt{a=ar+1} \Longrightarrow \texttt{P(ar)},\\ \texttt{P(a)} \wedge \texttt{a=0} \wedge \texttt{b=0} \Longrightarrow \texttt{Q(a,b)},\\ \texttt{P(a)}\wedge\texttt{an=a-1}\wedge\texttt{bn=b-1}\wedge\texttt{Q(an,bn)} \Longrightarrow \texttt{Q(a,b)}.\end{array}$$

Using suitable fix-point analysis techniques, we can synthesize the approximations which would add a pure post-condition `b=a` for the size properties. More specifically, we have $\texttt{P(a)} \equiv \texttt{true}$, $\texttt{Q(a,b)} \equiv \texttt{a=b} \wedge \texttt{b}\geq\texttt{0}$ and a new specification that ensures memory safety:

```
requires sllN(x,a)
ensures  dllN(x,q,b) ∧ b=a ∧ b≥0;
```

1.4.2 Second-Order Entailment Procedure

The bi-abductive entailment formalism in HIP system is of the following form

$$[v_1, .., v_n]\ \Delta_1 \vdash \Delta_2 \rightsquigarrow (\phi_p, \Delta_r, \mathcal{R})$$

where the left-hand side of $\rightsquigarrow$ is its input and the right-hand side is the output (Δ_r is the residua heap). The procedure infers the output s.t. the following entailment holds:

$$\mathcal{R} \wedge \Delta_1 \wedge \phi_p \vDash \Delta_2 * \Delta_r$$

Three new features are added here to support incremental inference:

- We specify a set of variables $[v_1, .., v_n]$ for which inference is to be applied. When the list is empty, the entailment system reduces to forward verification without any inference capability.
- We allow *second-order variables*, in the form of *uninterpreted* relations, to support incremental inference for pre/post specifications.
- We then collect a set of constraints captured by either ϕ_p (for the first-order selective variables) and a set of relational assumptions $\mathcal{R}$ (for the second-order variables) of the form $\mathcal{R} = \bigcup_{i=1}^{n}(\Delta_i \mid \Delta_g \Rightarrow \Delta'_i)$. $\mathcal{R}$ provides a set of interpretations for second-order variables. and can also be represented by a conjunction of the inferred constraints.

Shape Inference. There are two scenarios to consider for unknown predicates: (1) Δ_1 contains an *unknown* predicate instance that matched with a points-to or known predicate in Δ_2; (2) Δ_2 contains an *unknown* predicate instance.

An example of the first scenario is (where the data structure `snode` is defined as `data snode { snode next}`):

$$[\mathtt{U}]\ \mathtt{U(x)} \vdash \mathtt{x}{\mapsto}\mathtt{snode}\langle\mathtt{n}\rangle \rightsquigarrow (\mathtt{U(x)}{\Rightarrow}\mathtt{x}{\mapsto}\mathtt{snode}\langle\mathtt{n}\rangle{*}\mathtt{U_0(n)}, \mathtt{true},\ \mathtt{U_0(n)})$$

Here, we generated a relational assumption to denote an *unfolding* (or instantiation) for the unknown predicate `U` to a heap node `snode` followed by another unknown $\mathtt{U_0(n)}$ predicate. An example of the second scenario is shown next.

$$\begin{array}{l}[\mathtt{U_1}]\ \mathtt{x}{\mapsto}\mathtt{snode}\langle\mathtt{null}\rangle{*}\mathtt{y}{\mapsto}\mathtt{snode}\langle\mathtt{null}\rangle \vdash \mathtt{U_1(x)}\\ \quad\rightsquigarrow (\mathtt{true}, \mathtt{x}{\mapsto}\mathtt{snode}\langle\mathtt{null}\rangle{\Rightarrow}\mathtt{U_1(x)}, \mathtt{y}{\mapsto}\mathtt{snode}\langle\mathtt{null}\rangle)\end{array}$$

The generated relational assumption depicts a *folding* process for unknown $\mathtt{U_1(x)}$ which captures a heap state traversed from the pointer `x`. Both folding and unfolding of unknown predicates are crucial for second-order bi-abduction.

Bi-abductive unfold and fold are formalized in Fig. 1.7. For bi-abductive unfold, $\triangledown(\bar{w}, \pi)$ is an auxiliary function that existentially quantifies in π all free variables that are not in the set $\bar{w}$. Thus it eliminates from π all subformulas not related to $\bar{w}$ (*e.g.* $\triangledown(\{x, q\}, q{=}\mathtt{null}{\wedge}y{>}3)$ returns $q{=}\mathtt{null}$). An RHS assertion is either a points-to assertion $\mathtt{r}{\mapsto}\mathtt{c}\langle\mathtt{p}\rangle$ or a known predicate instance $\mathtt{P(r, p)}$ is paired through the parameter `r` with the unknown predicate `U`. Second, the unknown predicates $\mathtt{U_j}$ are generated for the data fields of κ_s. Third, the unknown predicate $\mathtt{U_{rem}}$ is generated for the instantiatable parameters $\mathtt{v_i}$ of `U`. The fourth and fifth lines compute relevant pure formulas and generate the assumption, respectively. Finally, the unknown predicates $\kappa_{\mathtt{fields}}$ and $\kappa_{\mathtt{rem}}$ are combined in the residue of LHS to continue discharging the remaining formula in RHS.

For bi-abductive fold, the function $\mathtt{reach}(\mathtt{w}, \kappa_1{\wedge}\pi_1, \mathtt{z\#})$ extracts portions from the antecedent heap (κ_1) that are (1) unknown predicates containing at least one instantiatable parameter from `w`; or (2) points-to or known predicates reachable from `w`, but not reachable from `z`. The $\mathtt{heaps}(\Delta)$ function enumerates all known predicate instances (of the form $\mathtt{P(v)}$) and points-to instances (of the form $\mathtt{r}{\mapsto}\mathtt{c}\langle\mathtt{v}\rangle$) in Δ. The function $root(\kappa)$ is defined as: $root(\mathtt{r}{\mapsto}\mathtt{c}\langle\mathtt{v}\rangle){=}\{\mathtt{r}\}$, $root(\mathtt{P(r, v)}) = \{\mathtt{r}\}$. In the first line, heaps of LHS are separated into the assumption κ_{11} and the residue κ_{12}. Second, heap guards (and their root pointers)

$$\frac{\begin{array}{c}[\underline{\text{INF-UNFOLD}}]\\ \kappa_{\mathtt{s}} \equiv \mathtt{r}{\mapsto}\mathtt{c}\langle\mathtt{p}\rangle \text{ or } \kappa_{\mathtt{s}} \equiv \mathtt{P}(\mathtt{r},\mathtt{p})\\ \kappa_{\mathtt{fields}} = *_{\mathtt{p_j}\in\mathtt{p}}\, \mathtt{U_j}(\mathtt{p_j},\mathtt{v_i}\#,\mathtt{v_n}\#) \text{ where } \mathtt{U_j}\text{: fresh preds}\\ \kappa_{\mathtt{rem}} = \mathtt{U_{rem}}(\mathtt{v_i},\mathtt{v_n}\#,\mathtt{r}\#) \text{ where } \mathtt{U_{rem}}\text{: a fresh pred}\\ \pi_{\mathtt{a}} = \triangledown(\{\mathtt{r},\mathtt{v_i},\mathtt{v_n},\mathtt{p}\},\pi_1) \quad \pi_{\mathtt{c}} = \triangledown(\{\mathtt{p}\},\pi_2)\\ \mathcal{A} \equiv (\mathtt{U}(\mathtt{r},\mathtt{v_i},\mathtt{v_n}\#) \wedge \pi_{\mathtt{a}} \Rightarrow \kappa_{\mathtt{s}} * \kappa_{\mathtt{fields}} * \kappa_{\mathtt{rem}} \wedge \pi_{\mathtt{c}})\\ [v^*]\ \kappa_1 * \kappa_{fields} * \kappa_{rem} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathtt{true}, \mathcal{R}, \Delta_R)\end{array}}{[\mathtt{U}, v^*]\ \mathtt{U}(\mathtt{r},\mathtt{v_i},\mathtt{v_n}\#) * \kappa_1 \wedge \pi_1 \vdash \kappa_s * \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathtt{true}, \mathcal{A}\wedge\mathcal{R}, \Delta_R)}$$

$$\frac{\begin{array}{c}[\underline{\text{INF-FOLD}}]\\ \kappa_{11}{=}\mathtt{reach}(\mathtt{w}, \kappa_1 \wedge \pi_1, \mathtt{z}\#) \quad \exists \kappa_{12} \cdot \kappa_1{=}\kappa_{11} * \kappa_{12}\\ \kappa_{\mathtt{g}} = *\{\kappa \mid \kappa{\in}\mathtt{heaps}(\kappa_{12}) \wedge root(\kappa){\subseteq}\mathtt{z}\} \quad \mathtt{r}{=}\bigcup_{\kappa\in\kappa_{\mathtt{g}}} root(\kappa)\\ \mathcal{A} \equiv (\kappa_{11} \wedge \triangledown(\mathtt{w}, \pi_1) \Rightarrow \mathtt{U_c}(\mathtt{w}, \mathtt{z}\#) \mid \kappa_{\mathtt{g}} \wedge \triangledown(\mathtt{r}, \pi_1))\\ [v^*]\ \kappa_{12} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathtt{true}, \mathcal{R}, \Delta_R)\end{array}}{[\mathtt{U_c}, v^*]\ \kappa_1 \wedge \pi_1 \vdash \mathtt{U_c}(\mathtt{w}, \mathtt{z}\#) * \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathtt{true}, \mathcal{A}\wedge\mathcal{R}, \Delta_R)}$$

Fig. 1.7 Bi-Abductive Unfolding and Folding.

$$\frac{[\underline{\text{INF-AND}}]\quad [v^*]\ \pi_1 \vdash \pi_2 \rightsquigarrow (\phi_2, \Delta_2, \mathcal{R}_2) \qquad [v^*]\ \pi_1 \vdash \pi_3 \rightsquigarrow (\phi_3, \Delta_3, \mathcal{R}_3)}{[v^*]\ \pi_1 \vdash \pi_2 \wedge \pi_3 \rightsquigarrow (\phi_2 \wedge \phi_3, \Delta_2 \wedge \Delta_3, \mathcal{R}_2 \cup \mathcal{R}_3)}$$

$$\frac{[\underline{\text{INF-UNSAT}}]\quad \text{UNSAT}(\alpha_1)}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\mathtt{true}, \mathtt{false}, \emptyset)} \qquad \frac{[\underline{\text{INF-VALID}}]\quad \alpha_1 \Rightarrow\ \alpha_2}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\mathtt{true}, \alpha_1, \emptyset)}$$

$$\frac{[\underline{\text{INF-LHS-CONTRA}}]\quad \phi = \forall(FV(\alpha_1){-}v^*) \cdot \neg\alpha_1 \quad \text{UNSAT}(\alpha_1 \wedge \alpha_2) \quad \phi{\neq}\mathtt{false}}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi, \mathtt{false}, \emptyset)} \qquad \frac{[\underline{\text{INF-PRE-DERIVE}}]\quad \phi{=}\forall(FV(\alpha_1, \alpha_2){-}v^*) \cdot (\neg\alpha_1 \vee \alpha_2) \quad \phi{\neq}\mathtt{false}}{[v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi, \alpha_1 \wedge \phi, \emptyset)}$$

$$\frac{[\underline{\text{INF-REL-DEFN}}]}{[v^*, v_{rel}]\ \pi \vdash v_{rel}(u^*) \rightsquigarrow (\mathtt{true}, \mathtt{true}, \{\pi{\rightarrow}v_{rel}(u^*)\})}$$

$$\frac{[\underline{\text{INF-REL-OBLG}}]\quad [u^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi_1, \Delta_1, \emptyset) \qquad [v^*]\ \alpha_1 \vdash \alpha_2 \rightsquigarrow (\phi_2, \Delta_2, \emptyset)}{[v^*, v_{rel}]\ \alpha_1 \wedge v_{rel}(u^*) \vdash \alpha_2 \rightsquigarrow (\phi_2, \Delta_1 \wedge \Delta_2, \{v_{rel}(u^*){\rightarrow}\phi_1\})}$$

Fig. 1.8 Pure Bi-Abduction Rules

are inferred based on κ_{12} and the #-annotated parameters $\mathtt{z}$. The assumption is generated in the third line and finally, the residual heap is used to discharge the remaining heaps of RHS.

Pure Inference. When both the antecedent and the consequent are heap free, the rules in Figure 1.8 for pure inference can apply. Take note that these rules are to be applied in a *top-down* and *left-to-right* order.

- INF-[AND] repeatedly breaks the conjunctive consequent into smaller components.
- INF-[UNSAT] and INF-[VALID] infer `true` precondition whenever the entailment already succeeds. Specifically, the rule INF-[UNSAT] applies when the antecedent α_1 of the entailment is unsatisfiable, whereas the rule INF-[VALID] is used if INF-[UNSAT] cannot be applied, meaning that the antecedent is satisfiable.

- The pure precondition inference is captured by two rules INF−[LHS−CONTRA] and INF−[PRE−DERIVE]. While the first rule handles antecedent contradiction, the second one infers the missing information from the antecedent required for proving the consequent. Specifically, whenever a contradiction is detected between the antecedent α_1 and the consequent α_2, then the rule INF−[LHS−CONTRA] applies and the precondition $\forall(FV(\alpha_1)-v^*)\cdot\neg\alpha_1$ contradicting with the antecedent is being inferred. Note that $FV(\cdots)$ returns the set of free variables from its argument(s), while v^* is a shorthand notation for $v_1,..,v_n$[1]. On the other hand, if no contradiction is detected, then the rule INF−[PRE−DERIVE] infers a sufficient precondition required for proving the consequent.
- The last two rules INF−[REL−DEFN] and INF−[REL−OBLG] are meant to gather definitions and obligations, respectively, for the uninterpreted relation $v_{rel}(u^*)$. For simplicity, in the rule INF−[REL−OBLG], we just formalize the case when there is only one uninterpreted relation in the antecedent.

For example, to illustrate how the selective inference works, consider three entailments below with $\texttt{x}{\mapsto}\texttt{node}\langle\_,\texttt{q}\rangle$ as a consequent:

$$[\texttt{n}]\ \texttt{sllN}\langle\texttt{x},\texttt{n}\rangle \vdash \texttt{x}{\mapsto}\texttt{node}\langle\_,\texttt{q}\rangle \rightsquigarrow (\texttt{n}{>}\texttt{0}, \texttt{sllN}\langle\texttt{q},\texttt{n}{-}1\rangle, \texttt{true})$$
$$[\texttt{x}]\ \texttt{sllN}\langle\texttt{x},\texttt{n}\rangle \vdash \texttt{x}{\mapsto}\texttt{node}\langle\_,\texttt{q}\rangle \rightsquigarrow (\texttt{x}{\neq}\texttt{null}, \texttt{sllN}\langle\texttt{q},\texttt{n}{-}1\rangle, \texttt{true})$$
$$[\texttt{n},\texttt{x}]\ \texttt{sllN}\langle\texttt{x},\texttt{n}\rangle \vdash \texttt{x}{\mapsto}\texttt{node}\langle\_,\texttt{q}\rangle \rightsquigarrow (\texttt{n}{>}\texttt{0}{\vee}\texttt{x}{\neq}\texttt{null}, \texttt{sllN}\langle\texttt{q},\texttt{n}{-}1\rangle, \texttt{true})$$

Predicate $\texttt{sllN}\langle\texttt{x},\texttt{n}\rangle$ by itself does not entail a non-empty node. For the entailment checking to succeed, the current state would have to be strengthened with either $\texttt{x}{\neq}\texttt{null}$ or $\texttt{n}{>}\texttt{0}$. Our procedure can decide on which pre-condition to return, depending on the set of variables for which pre-conditions are to be built from. The selectivity is important since we only consider a subset of variables (e.g. $\texttt{a},\texttt{b},\texttt{r}$), which are introduced to capture pure properties of data structures.

1.4.3 Specification Inference via Hoare-Style Rules

To support specification inference via second-order bi-abduction, we extend the Hoare-style forward rule presented in Section 1.2 to the form: $[v^*] \vdash \{\Delta_1\}\ c\ \{\phi_2, \Delta_2, \mathcal{R}_2\}$ with three additional features (i) a set of specified variables $[v^*]$ (ii) an extra precondition ϕ_2 that must be added (iii) a set of definitions and obligations $\mathcal{R}_2$ on the uninterpreted relations. The selectivity criterion will help ensure that ϕ_2 and $\mathcal{R}_2$ come from only the specified set of variables, namely $\{v^*\}$. If this set is empty, our new rule is simply a special case that only performs verification, without any inference.

Figure 1.9 captures a subset of our Hoare rules with bi-abduction. For each method call, we must ensure that its precondition is satisfied, and then add the expected postcondition into its residual state, as illustrated in [INF−METH−CALL]. Here, $(t_i\ v_i)_{i=1}^{m-1}$ are pass-by-reference parameters, which are marked with `ref`, while the pass-by-value parameters V are equated to their initial values through the *nochange* function, as their updated values are not visible in the method's callers. Note that post-state $\Phi_{\texttt{po}}$ captures updates that may

[1] If there is no ambiguity, we can use v^* instead of $\{v^*\}$.

$$\frac{\begin{array}{c}[\text{INF-METH-CALL}]\\ t_0\ mn\ (\texttt{ref}\ (t_i\ v_i)_{i=1}^{m-1}, (t_j\ v_j)_{j=m}^{n})\ \ \Phi_{pr}\ \ \Phi_{po}\ \{c\} \in Prog \\ \rho{=}[v'_k/v_k]_{k=1}^{n} \quad \Phi'_{pr} = \rho(\Phi_{pr}) \qquad W{=}\{v_1,\ldots,v_{m-1}\} \quad V{=}\{v_m,\ldots,v_n\} \\ [v^*]\ \Delta \vdash \Phi'_{pr} \rightsquigarrow (\phi_2, \Delta_2, \mathcal{R}_2) \qquad \Delta_3{=}(\Delta_2 \wedge nochange(V))\ *_W \Phi_{po}\end{array}}{[v^*] \vdash \{\Delta\}\ mn(v_1,\ldots,v_{m-1},v_m,\ldots v_n)\ \{\phi_2, \Delta_3, \mathcal{R}_2\}}$$

$$\frac{\begin{array}{c}[\text{INF-METH-DEF}]\\ [v^*, v^*_{rel}] \vdash \{\Phi_{pr} \wedge \bigwedge(u'{=}u)^*\}\ c\ \{\phi_2, \Delta_2, \mathcal{R}_2\} \qquad [v^*, v^*_{rel}]\ \Delta_2 \vdash \Phi_{po} \rightsquigarrow (\phi_3, \Delta_3, \mathcal{R}_3) \\ \rho_1 = infer\_pre(\mathcal{R}_2 \cup \mathcal{R}_3) \qquad \rho_2 = infer\_post(\mathcal{R}_2 \cup \mathcal{R}_3) \\ \Phi^n_{pr} = \rho_1(\Phi_{pr} \wedge \phi_2 \wedge \phi_3) \qquad \Phi^n_{po} = \rho_2(\Phi_{po} * \Delta_3)\end{array}}{\vdash\ t_0\ mn\ ((t\ u)^*)\ \texttt{infer}\,[v^*, v^*_{rel}]\ \Phi_{pr}\ \Phi_{po}\ \{c\} \rightsquigarrow \Phi^n_{pr}\ \Phi^n_{po}}$$

Fig. 1.9 Hoare Rules with Bi-Abduction

occur from pass-by-ref parameters `W`. These updates need to be appropriately linked to caller's pre-stage Δ_2 via the compose-with-update operator $*_W$. Note that inference is allowed to occur during the entailment of the method's precondition.

Lastly, we discuss the rule for handling each method declaration [INF-METH-DEF]. At the program level, our inference rules will be applied to each set of mutually-recursive methods in a bottom-up order in accordance with the call hierarchy. This allows us to gather the entire set $\mathcal{R}$ of relational assumptions for each uninterpreted relation. For inferring each shape-based relation, we make use of the algorithm presented in [53] to derive a definition for it. Our algorithm uses derivation rules that are sound but is currently incomplete since heuristics are deployed to make the approach practical. For a pure relation, we infer the pre- and post-relations via the two steps described below. Take note that, given the entire set $\mathcal{R}$, we retrieve the set of definitions and obligations for post-relations through functions $\texttt{def}_{\texttt{po}}$ and $\texttt{obl}_{\texttt{po}}$ respectively, while we use functions $\texttt{def}_{\texttt{pr}}$ and $\texttt{obl}_{\texttt{pr}}$ for pre-relations. Pre-relations denote unknown relations that are used in the pre-condition, while post-relation denote those unknown relations that are being used in the post-conditions. We also used annotations `@pr` and `@po` to explicitly identify the pre-relations and post-relations, respectively.

$$\begin{array}{l}\texttt{def}_{\texttt{po}}(\mathcal{R}) = \{\pi^k_i \rightarrow v_{rel_i}(v^*_i) \mid (\pi^k_i \rightarrow v_{rel_i}@\texttt{po}(v^*_i)) \in \mathcal{R}\}\\ \texttt{obl}_{\texttt{po}}(\mathcal{R}) = \{v_{rel_i}(v^*_i) \rightarrow \alpha_j \mid (v_{rel_i}@\texttt{po}(v^*_i) \rightarrow \alpha_j) \in \mathcal{R}\}\\ \texttt{def}_{\texttt{pr}}(\mathcal{R}) = \{\pi^k_i \rightarrow v_{rel_i}(v^*_i) \mid (\pi^k_i \rightarrow v_{rel_i}@\texttt{pr}(v^*_i)) \in \mathcal{R}\}\\ \texttt{obl}_{\texttt{pr}}(\mathcal{R}) = \{v_{rel_i}(v^*_i) \rightarrow \alpha_j \mid (v_{rel_i}@\texttt{pr}(v^*_i) \rightarrow \alpha_j) \in \mathcal{R}\}\end{array}$$

- In order to infer post-relations, the function *infer_post* applies a least fixed point analysis over the sets $\texttt{def}_{\texttt{po}}(\mathcal{R})$ and $\texttt{obl}_{\texttt{po}}(\mathcal{R})$. For computing the least fixed point in the two domains used in the current inference framework, namely the numerical domain and the set/bag domain, we utilize FixCalc [79] and FixBag [77], respectively.
- For pre-relations, our goal is to infer the weakest preconditions via *infer_pre*. Hence, for each pre-relation, we first calculate the conjunction of all its obligations from $\texttt{obl}_{\texttt{pr}}(\mathcal{R})$ to obtain sufficient preconditions for base cases. To capture the precondition

for a recursive call, we need to derive the *recursive invariant* which can be achieved via a top-down fixed point analysis [80].
Through the functions *infer_pre* and *infer_post*, we can finally infer definitions of uninterpreted relations for deriving the pre- and postconditions, Φ^n_{pr} and Φ^n_{po}, of a method *mn*. Note that v^*_{rel} denotes the set of uninterpreted relations that are to be inferred, whereas ρ_1 and ρ_2 represent the substitutions obtained for pre- and post-relations, respectively.

1.5 Termination and Non-Termination Reasoning

The problems of proving program *termination* and *non-termination* are orthogonal. While termination can be encoded as a *liveness* property, non-termination is considered as a *safety* property. In this section, we will introduce a *unified* component in the HIP system for reasoning *both* termination and non-termination of imperative programs at the same time. The component, whose structure was represented in [60], was implemented based on two main techniques:

- A resource-based specification logic and an automated verification system for specifying and verifying termination and non-termination properties of programs [58].
- A modular inference system for automatically inferring termination and non-termination specification of the programs. The system employs an abductive inference technique with second-order specification to derive a summary of terminating and non-terminating behaviors for each method in the program [59].

1.5.1 A Verification System for Termination and Non-termination

We propose three primitive temporal predicates `Term` X with the ranking function X, `Loop`, and `MayLoop` to specify the program termination, definite non-termination, and possibly non-termination, respectively. We then extend the core specification language of HIP (cf. Section 1.2.2) to these temporal predicates and integrate their reasoning into HIP's forward verification system (cf. Section 1.2.3) to specify and verify program termination and non-termination. As a result, we can utilize the rich specification language and the available verification infrastructure for reasoning about terminating and non-terminating behaviors of various programs.

To do that, we propose a logic with the unified predicate $\texttt{RC}\langle l, u\rangle$ denoting the lower bound l and the upper bound u of available resource capacity for program execution, given that $0 \leq l \leq u$. In this logic, each method requires an initial resource for its execution, specified by a predicate $\texttt{RC}\langle l, u\rangle$ in its precondition, and the verification system then statically monitors the consumption of this resource to guarantee that it is always sufficient for the execution. To keep track of the consumed and remaining resources within the same entailment checking of the existing verification system, we design an entailment checking with *frame* for resource reasoning via the resource splitting operation $\blacktriangleright$, which is similar

to the heap separating conjunction in separation logic. In particular, the splitting resource assertion $\mathtt{RC}\langle l_1, u_1\rangle \blacktriangleright \mathtt{RC}\langle l_2, u_2\rangle$ holds for a resource indicated by $\mathtt{RC}\langle l, u\rangle$ if and only if that resource can be split into two resource fragments, on which $\mathtt{RC}\langle l_1, u_1\rangle$ and $\mathtt{RC}\langle l_2, u_2\rangle$ hold respectively. In this case, we say that the entailment $\mathtt{RC}\langle l, u\rangle \vdash \mathtt{RC}\langle l_1, u_1\rangle \blacktriangleright \mathtt{RC}\langle l_2, u_2\rangle$ is valid. Intuitively, this entailment encodes a resource consumption of $\mathtt{RC}\langle l_1, u_1\rangle$ on the original resource $\mathtt{RC}\langle l, u\rangle$ with the remaining resource $\mathtt{RC}\langle l_2, u_2\rangle$. The entailment checking ensures that $\mathtt{RC}\langle l, u\rangle$ is sufficient for $\mathtt{RC}\langle l_1, u_1\rangle$ to be consumed. During program verification, the resource entailment checking mainly happens at the method calls, so that we can check if the current maximum resource of the caller satisfies the maximum resource requirement needed by the callee. In addition, at the end of each method declaration, we check if the lower bound of the remaining resource is equal 0 in order to ensure that the minimum resource utilized is suitably verified for each method declaration.

Specifically for the termination and non-termination reasoning, we model the three temporal predicates as follows:

$$\begin{aligned} \mathtt{Term}\ X &\triangleq \mathtt{RC}\langle 0, \rho(X)\rangle \\ \mathtt{Loop} &\triangleq \mathtt{RC}\langle \infty, \infty\rangle \\ \mathtt{MayLoop} &\triangleq \mathtt{RC}\langle 0, \infty\rangle \end{aligned}$$

where $\rho(X)$ is an order-embedding of the ranking function X into naturals. Intuitively, a terminating method indicated by $\mathtt{Term}\ X$ must have a finite upper bound of the resource to execute and $\mathtt{0}$ lower bound. On the other hand, a definitely non-terminating method indicated by $\mathtt{Loop}$ requires an infinite lower bound of the resource, so that it will ensure infinite execution/consumption. Lastly, a possibly non-terminating method indicated by $\mathtt{MayLoop}$ does not have any specific requirement, so that its required resource ranges from 0 to ∞. The resource entailment checking rules of these specific predicates are:

$$\begin{array}{ll} \mathtt{MayLoop} \vdash \mathtt{MayLoop} \blacktriangleright \mathtt{MayLoop} & \mathtt{Loop} \vdash \mathtt{Loop} \blacktriangleright \mathtt{MayLoop} \\ \mathtt{MayLoop} \vdash \mathtt{Loop} \blacktriangleright \mathtt{MayLoop} & \mathtt{Loop} \vdash \mathtt{Term}\ X \blacktriangleright \mathtt{Loop} \\ \mathtt{MayLoop} \vdash \mathtt{Term}\ X \blacktriangleright \mathtt{MayLoop} & \dfrac{Y < X}{\mathtt{Term}\ X \vdash \mathtt{Term}\ Y \blacktriangleright \mathtt{Term}\ X} \\ \mathtt{Loop} \vdash \mathtt{MayLoop} \blacktriangleright \mathtt{Loop} & \end{array}$$

These rules follow the general resource entailment checking for monitoring the execution length of the method. A program state with $\mathtt{MayLoop}$ or $\mathtt{Loop}$ accepts any other resource requirement by their ∞ upper bound. However, the ∞ lower bound of $\mathtt{Loop}$ is only consumed by the other $\mathtt{Loop}$ to become a $\mathtt{MayLoop}$. A program state with $\mathtt{Term}\ X$ only accepts another program state with a finite resource requirement $\mathtt{Term}\ Y$ which is smaller than X. Here, the frame $\mathtt{Term}\ X$ is an over-approximation of the actual remaining resource. This over-approximation is safe since its upper bound is finite and does not exceed the original resource requirement. The other entailments, such as $\mathtt{Term}\ X \vdash \mathtt{Loop} \blacktriangleright \_$ and $\mathtt{Term}\ X \vdash \mathtt{MayLoop} \blacktriangleright \_$, are invalid since they indicates that the caller's resource does not meet the callee's resource requirement. Finally, the termination and non-termination reasoning follows the resource reasoning by requiring that $\mathtt{Loop}$ resource state, with a non-zero lower bound, are never encountered at the end of each method declaration, since that would indicate some unconsumed resource lower bounds by the body of a method that had already been earlier declared as definitely non-terminating.

1.5.2 A Termination and Non-termination Specification Inference

In the next step, we aim to automatically infer the termination specifications with `Term` X, `Loop`, and `MayLoop` associated with their pre-conditions. The general idea is that given a pre-condition (`true` initially), we attempt to prove if under this pre-condition, the method terminates (by determining if the pre-condition is a base case or otherwise synthesizing a valid ranking function) or does not terminate (by proving that the method exit is unreachable). The details of this process is captured in [59] and will also introduce and infer unknown predicates to capture the expected lower and upper resource bounds. If resource bound inference fail, we apply an *abductive inference* on the failure of the non-termination proof to derive a new pre-condition that focus on the non base case(s) in the *next step* of the execution.

Using *case analysis*, we refine the currently considered pre-condition into two distinct cases: the newly derived abductive pre-condition and its complement. We then repeat the inference process with these two new pre-conditions. The refinement iteration stops when we can determine the termination (`Term` X) or non-termination (`Loop`) status of all pre-conditions, so that there is no new unknown pre-conditions derived or when the number of iterations reaches a preset number, in which all unknown pre-conditions are marked as `MayLoop`.

Note that we do not implement the termination and non-termination inference from scratch but rather utilize the existing HIP verification and specification inference system. We must of course extend our specification logic to support unknown temporal predicate; in addition to the three existing known temporal predicates `Term` X, `Loop`, and `MayLoop` and then infer solutions for each new unknown predicate, in terms of the known predicates.

Then, we annotate the analyzed method with an unknown specification and use the same termination and non-termination reasoning. Our specification inference step would again collect a set of relational assumptions over the unknown predicates. These assumptions can be subjected to a refinement iteration process to obtain possible solutions of the unknown predicates. By not analysing the source programs directly but subjecting them to a verification against specifications containing unknowns, our algorithm can also be referred to as a *second-order* inference technique.

1.6 Implementation and Experiments

HIP/SLEEK is written in OCaml consisting of approximately 260 source files, with approximately 232K lines of code. It is structured in a modular manner to facilitate the addition of new features, allowing it to become a more fully featured software verifier.

HIP and SLEEK work together in tandem, with HIP generating proof obligations on code that are to be proven by SLEEK. By default, HIP works on a C-like language that allows for the definition of data structures, shape predicates involving these data structures, and function pre- and post-conditions. In addition, it also has a variety of extended language parsers that work on common languages like C and Java, which allow

the specification of predicates and pre- and post-conditions inside specially annotated comment blocks. This allows it to work on some real world programs.

SLEEK works with a syntax that closely mimics standard mathematical notation for logic, with some notational simplifications for use as code (such as & for $\wedge$). SLEEK's language also includes similar syntax to HIP for defining data structures and shape predicates, which allows it to be used independently of HIP as an entailment prover that can handle user-defined data structures and predicates.

Entailment in SLEEK proceeds by matching up heap nodes from the antecedent to the consequent, following the rules described in Section 1.3. Once the consequent only contains pure formulae, we use $\texttt{XPure}_n$ to soundly approximate any remaining heap formula in the antecedent as a pure formula. SLEEK then uses a range of theorem provers to discharge the now pure entailment. The theorem provers include Z3 [66], MONA [28], Isabelle [72], the Coq proof assistant or the Omega Calculator constraint solver [81]. Z3 is used by default, with the other provers being enabled by the user as needed.

Note that as each theorem prover has different syntax for the input formulas, SLEEK needs to transform formulae into the proper format before invoking it. For example, the Omega Calculator does not support the *max* function, e.g., the formula $z = max(x, y)$, directly. Thus, before using Omega Calculator to discharge $z = max(x, y)$, SLEEK transforms the formula into the equivalent one $z = y \wedge y > x \vee z = x \wedge x \geq y$ which is accepted by Omega Calculator

1.6.1 Proof Search Heuristic

In contrast to most entailment provers, SLEEK employs a relatively straightforward proof search heuristic when attempting to prove an entailment. If it sees a data node on the left hand side, and another on the right, with both having the same name, it will attempt to match both nodes, and remove them from the next entailment step. For a pair of predicate instances, we will attempt a match, followed by a fold if it does not succeed, followed by an unfold if the fold fails.

For a data node on the left, and a view on the right, we attempt fo fold the data node with other nodes on the left to try to obtain the same view that we have on the right. In the reverse situation (i.e. view on the left, data on the right), we unfold the view and attempt to match the new nodes.

If lemmas (Sec. 1.3) are involved, we expand the search to include the applicable lemmas. These lemmas are applied where possible, and the proof search continues with the entailments that were transformed with the lemmas.

1.6.2 Experiments

Table 1.1 shows the experimental results for a suite of test programs over a variety of data structures. The tests were performed on an Intel® Core™ i7-960, 3.20 GHz. For each example, we note the following information (in order):

- The function being verified.
- The number of lines of code of the function. This includes the lines of code for all functions that are being called by the function being verified.
- The number of lines of annotations needed, including shape definitions.
- The verification time, when Z3 and the Omega Calculator are used to discharge pure obligations.
- The verification time, when MONA is used to discharge pure obligations.

We also note the additional properties verified beside each category. The verification times are given in seconds, to 3 significant figures.

The average cost of annotation, i.e. the number of lines of annotations to the number of lines of code, is 18

Table 1.1 Verification times (in seconds) for data structures with arithmetic and bag/set constraints

Program	Lines of code	Lines of annotation	Verification Time		Verification Time
			Z3 + Omega	MONA	MONA
Linked List			*size/length*		*bag/set*
delete	11	5	0.712	1.03	1.90
reverse	11	5	0.726	1.08	2.40
Circular List			*size + cyclic structure*		*bag/set + cyclic structure*
delete (first)	11	5	0.568	0.723	2.53
count	24	10	0.611	0.714	2.52
Doubly-Linked List			*size + double links*		*bag/set + double links*
append	20	5	1.02	2.11	3.22
delete	19	5	1.28	3.45	29.69
Sorted List			*size + min + max + sortedness*		*bag/set + sortedness*
delete	18	5	0.911	20.8	3.41
insertion_sort	30	10	1.01	failed	2.58
selection_sort	39	13	0.691	5.24	2.11
bubble_sort	31	17	0.533	0.556	1.34
merge_sort	87	17	1.09	failed	12.1
quick_sort	63	17	2.82	failed	6.69
Binary Search Tree			*min + max + sortedness*		*bag/set + sortedness*
insert	21	5	1.31	25.0	5.20
delete	48	7	1.39	20.8	10.2
AVL Tree			*size + height + height-balanced*		*bag/set + height + height-balanced*
insert	118	20	31.2	failed	failed
delete	129	20	10.92	11.2	31.3
Red-Black Tree			*size + height + height-balanced*		*bag/set + height + height-balanced*
insert	236	50	8.71	failed	106
delete	308	50	2.89	2.97	165

We give a brief summary of the properties captured in each category:

- For singly-linked lists (**Linked List**), **circular lists**, and **doubly-linked lists**, the specifications capture the size of the lists (i.e. the total number of nodes). For circular

and doubly-linked lists, the cyclic structure and the double links, respectively, are captured as well.
- For **sorted lists**, the size of the list, the minimum element, and maximum elements are tracked. The sortedness property is expressed using the minimum element. If the specifications contain the bag or set of reachable values, the sortedness is expressed directly over the bag or set, with no need to explicitly track the minimum value.
- **Binary search trees** require the tree elements to be sorted. As such, similar to sorted lists, we capture this sortedness property by either tracking the minimum/maximum values within the tree, or via the bag or set of reachable values.
- For **AVL trees**, we capture the total number of nodes in the tree (its *size*), and its height. Additionally, we specify and invariant that ensures that the tree is *height-balanced*, i.e. that its left and right subtrees are nearly balanced. Even when we have the bag or set of reachable values, we continue to track the height of the tree, to maintain this invariant.
- For **red-black trees**, we track the size and the *black height* (i.e. the height when considering only the black nodes). We also include an invariant that ensures that the tree is height-balanced in its black height, meaning that each node's left and right subtree have the same black height.

By default, we utilise a combination of Z3 and the Omega Calculator to discharge pure proof obligations, with each prover covering for the weaknesses of the other. The effectiveness of this combination is seen in the speed at which the tests complete, with Z3 and the Omega Calculator verifying the examples faster than MONA, and sometimes an order of magnitude faster.

However, this combination of Z3 and the Omega Calculator is limited in its ability to handle sets and bags of values, and thus when specifications involve bags or sets, we turn to MONA to discharge these proof obligations. MONA is, however, also limited in its ability to handle complex formulae. When the formula becomes too complex, MONA is unable to proceed with discharging the necessary pure proof obligations. This is noted where MONA fails to verify the example.

HIP/SLEEK has also participated in 2 software verification competitions over the years: the Software Verification Competition, or SV-COMP, and the Separation Logic Competition, or SL-COMP. Here, we present the results from SV-COMP 2016, and SL-COMP 2014.

Table 1.2 SV-COMP 2016 Results

Category	Total Tests	Correct	Incorrect	Time Taken (s)	Max Score	Score Obtained
Recursive	98	70	0	9700	151	111

In SV-COMP 2016, HIP participated in the Recursive category, which tests the ability of software verifiers to do recursive analysis on programs [2]. HIP's performance in SV-COMP 2016 is summarised in Table 1.2.

[2] `https://sv-comp.sosy-lab.org/2016/benchmarks.php`

Table 1.3 SL-COMP 2014 Results

Category	Total Tests	Correct	Incorrect	Time Taken (s)
sll(⊨)	110	110	0	4.99
sll(⇒)	292	292	0	14.13
FDB(⇒)	43	31	1	43.65
UDB(⊨)	61	61	0	30.84
UDB(⇒)	172	131	4	80.60

SLEEK also participated in SL-COMP 2014. The results are summarised in Table 1.3. The individual categories are[3]:

- sll(⊨): satisfiability problems for symbolic heaps with list segment predicates,
- sll(⇒): entailment problems for symbolic heaps with list segment predicates,
- FDB(⇒): entailment problems for symbolic heaps with composed lists,
- UDB(⊨): satisfiability problems for symbolic heaps with inductive definitions,
- UDB(⇒): entailment problems for symbolic heaps with inductive definitions.

HIP/SLEEK is available as an open source at [1] or through its web interface at [2].

1.7 Related Work

1.7.1 Formalisms for Shape Analysis of Data Structures

Many formalisms have been proposed for analyzing the shape of data structures in imperative programs. One well-known work is the Pointer Assertion Logic [65], by Moeller and Schwartzbach, which is a highly expressive mechanism to describe invariants of graph types [40]. The Pointer Assertion Logic Engine (PALE) uses Monadic Second-Order Logic over Strings and Trees as the underlying logic and the tool MONA [35] as the prover. PALE invariants are not designed to handle arithmetic, hence it is not possible to encode height-balanced trees in PALE. Moreover, PALE is unsound in handling procedure calls [65], whereas we would like to have a sound verifier.

Harwood et al. [34] describe a UTP theory for objects and sharing in languages like Java or C++. Their work focuses on a denotational model meant to provide a semantical foundation for refinement-based reasoning or Hoare-style axiomatic reasoning. Our work focuses more on practical verification for heap-manipulating programs.

In an object-oriented setting, the Dafny language [63] uses dynamic frames in its specifications. The term frame refers to a set of memory locations, and an expression denoting a frame is dynamic in the sense that as the program executes, the set of locations denoted by the frame can change. A dynamic frame is thus denoted by a set-valued expression (in particular, a set of object references), and this set is idiomatically stored in a field. Methods in Dafny use modifies and reads clauses, which frame the modifications

[3] from https://github.com/sl-comp/SL-COMP14/tree/master/bench

of methods and dependencies of functions. By comparison, separation logic provides a reasoning logic that hides the explicit representation of dynamic frames.

For shape inference, Sagiv et al. [85] present a parameterized framework, called TVLA, using 3-valued logic formulae and abstract interpretation. Based on the properties expected of data structures, the users may either learn or supply a set of predicates to the framework which are then used to analyse that certain shape invariants are maintained.

However, most of these techniques are focused on analysing shape invariants, and do not attempt to track the size and bag properties of complex data structures. An exception is the quantitative shape analysis [84] where a data flow analysis is proposed to compute quantitative information for programs with destructive updates. By tracking unique points-to reference and its height property, their algorithm is able to handle AVL-like tree structures. Even then, the author acknowledges the lack of a general specification mechanism for handling arbitrary shape/size properties.

1.7.2 Reasoning with Inductive Heap Predicates

In separation logic, the first automated procedure to handle inductive heap predicates was proposed by Berdine et al. [4, 5]. It reasons about the recursive structure of an inductive heap predicate by folding/unfolding the predicate against its definition. However, this work is hardwired to work for only `lseg` and `tree` predicates. Furthermore, it only performs predicate unfolding in the consequent of an entailment which may miss bindings on free variables. Compared to [4, 5], our unfold/fold mechanism is general, automatic and terminates for heap entailment checking.

Bi-abduction technique for compositional shape inferece was first introduced in [9], and was shown surprising effective and scalable to real program codes, despite its abstract domain being presently restricted to `lseg` predicate. Our specification inference process is inspired by this work, with the new goal of increasing the expressiveness of specifications that could be both inferrable and provable.

Besides [4, 5], to date, there are also various approaches that hard-wire inductive heap predicates to model specific types of the linked list and the tree data structures, such as the works of Piskac et al. [78], Bozga et al. [7], Perez et al. [74, 75], and Curry et al. [25]. These works provide specific syntax and semantics for predicates in advance so that they can derive efficient techniques to handle these predicates when proving entailments. Since the invented techniques are tied to certain types of pre-defined predicates, they might not be automatically extended to reason about other inductive heap predicates.

A more general approach is to consider classes of inductive heap predicates satisfying certain syntactic or semantic restrictions, such as predicates with a bounded tree width property by Iosif et al. [37, 38] or predicates describing variants of the linked list data structure by Enea et al. [29]. These authors propose to prove entailments by translating separation logic entailments into equivalent formulas in theories of automata or graphs. Thereby, they can employ developed proof techniques in automata and graph theories to prove the translated formulas, and conclude about the validity of the original separation logic entailments. Nevertheless, inductive heap predicates in this approach might not be able to represent sophisticated properties of data structures such as arithmetic constraints

about their size or elements' content. These constraints are not directly supported by the considered external theories of graphs and automata.

To resolve the above expressiveness limitation, in this work, we consider more general classes of user-defined inductive heap predicates, i.e. predicates which can be arbitrarily defined by users of verification and analysis systems and propose to prove entailments by using sequent-based proof systems with the folding/unfolding and predicate matching/removing mechanisms. Similar approaches are also utilized by Qiu et al. [82, 73], and Enea et al. [30]. However, there is a limitation in all of these works and ours: the proof derivation to unfold and match predicates can be infinite. Therefore, we sometimes require users to provide supplementing lemmas assisting the proof system to compose, decompose or reorganize inductive heap predicates without the need of unfolding.

In recent works, Brotherston et al. [8], Chu et al. [16], and Ta et al. [88, 90] propose to overcome the above limitation through inductive inference systems. Particularly, the infinite unfolding sequences of inductive heap predicates would be avoided by the detection of proof cycles [8, 54] or the application of induction hypotheses [16, 88, 90]. Finally, in [89], Ta et al. present a technique to automatically synthesize lemmas by combining induction proof and constraint solving, to assist proving entailments. Certainly, this work can free the users from the labor task of manually inspecting and providing necessary lemmas, which are needed in the proofs. Gillian, a reasoning platform that combines separation-logic based verification, bi-abductive compositional analysis and symbolic execution testing, tackles this problem by symbolically executing tests [32]. It explores all possible paths by unrolling loops up to a bound, concluding with a bounded verification guarantee in case of a successful verification or a counter-model, otherwise.

1.7.3 Beyond Shape: Size, Set, and Bag Properties

In another direction of research, size properties are mostly explored for declarative languages [36, 93, 13] as the immutability property makes their data structures easier to analyse statically. Size analysis is also extended to object-based programs [14] but is restricted to tracking either size-immutable objects that can be aliased and size-mutable objects that are unaliased, with no support for complex shapes.

The Applied Type System (ATS) [11] proposes combining programs with proofs. ATS capture program invariants using dependent types and are extremely expressive; it can express many program properties with the help of accompanying proofs. Using linear logic, ATS may also precisely handle mutable data structures with sharing. However, users must supply all expected properties, and precisely state where they are to be applied, with ATS playing the role of a proof-checker. In comparison, we use a more limited class of constraint for shape, size and bag analysis but support automated modular verification.

On the other hand, set-based analysis is proposed to verify data structure consistency properties in the work of Kuncak et al. [42], where a decision procedure is given for a first order theory that combines set and Presburger arithmetic. This result may be used to build a specialised mixed constraint solver but it currently has high algorithmic complexity. Lahiri and Qadeer [43] report an intra-procedural reachability analysis for well-founded

linked lists using first-order axiomatization. Reachability analysis is related to set/bag property that we capture but implemented by transitive closure at the predicate level.

1.7.4 Other Verification Systems in Non-Separation-Logic Settings

Program verifiers that are based on Hoare-style logic have been around longer than those based on separation logic. We describe some major efforts in this direction.

ESC/Java [31] is a verification system developed at Compaq Systems Research Center, that aims to detect more errors than traditional static checking tools, such as type checkers, but is not designed to be a program verification system. The stated goals of ESC/Java are scalability and usability. For that, it forgoes soundness for the potential benefits of more automation and faster verification time. Hence, ESC/Java suffers from both false negatives (programs that pass the check may still contain errors that ESC/Java is designed to handle), and false positives (programs flagged as erroneous are in fact correct programs). In contrast, our verifier is sound as it does not suffer from false negatives: if a program is verified, it is guaranteed to meet its specifications for all possible program executions.

ESC/Java2 [18] is a continual effort of ESC/Java which adds support for current versions of Java, and also verifies more JML [62] constructs. One significant addition is the support for model fields and method calls within annotations [17]. Since ESC/Java2 continues to use Simplify [27] as its underlying theorem prover which does not support transitive closure operations, it may have difficulties in verifying properties of heap-based data structures that require reachability properties, such as collections of values stored in container data structures.

Spec# [3] is a programming system developed at Microsoft Research to verify C# programms by utilizing its underlying verifier Boogie [3]. Spec# adds constructs tailored to program verification, such as pre- and post-conditions, frame conditions, non-null types, model fields and object invariants. Spec# also supports runtime assertion checking and object invariants. In order to verify invariants, Spec# employs an ownership scheme that allows an object to own its representation. This ownership scheme requires programmers to write special commands to enforce unpacking and packing objects' invariants. In our system, instead of using special fields in method contracts to indicate whether an invariant should be enforced, users directly use predicates. Hence, there is no need for explicitly packing and unpacking the objects in the method body. Consequently, users are shielded from the details of the verification methodology, which are largely irrelevant, from a user's point of view.

Jahob [41] is a verification system that mainly focuses on reasoning techniques for data structure verification that combines multiple theorem provers to reason about expressive logical formulas. Jahob uses a subset of the Isabelle/HOL [71] language as its specification language, and works on instantiatable data structures, as opposed to global data structures used in its predecessor, Hob [44]. Like SPEC# , Jahob supports ghost variables and specification assignments which places onus on programmers to help in the verification process by providing suitable instantiations of these specification variables.

Inspired by separation logic with higher order list predicates, Predator is an graph-based automated formal verification tool for verifying pointer manipulating C programs [76]. It

supports the verification of programs with pointer arithmetic and it distinguishes between safe and unsafe usage of invalid pointers. While Predator has a very mature treatment of programs manipulating singly or doubly linked lists, it only tackles trees and skip-lists in a restricted manner, that is for error detection and not for proving program safety.

EVE Proofs [92] is a verification system for Eiffel programs by translating and conducting the verification with Boogie [3]. To infer the frame condition, this tool relies on an automatic extraction of modifies clauses, which can be unsound. In contrast, our approach does not have to infer frame conditions, courtesy to the frame rule of separation logic [83]. Another restriction of EVE Proofs regards the methodology for invariants, which has to take into account that objects can temporarily violate the invariant, but also that an object can call other objects while being in an inconsistent state. As this is not considered at the moment, the current implementation of invariants can introduce unsoundness in the system.

For a comparison to the above non-separation-logic-based systems, our user-defined predicates, which capture the properties to be analysed, can remove the need for model fields and having object invariants tied to class/type declarations. Regarding ghost specification variables, they are not required since we provide support for automatically instantiating the predicates' parameters. Furthermore, we utilize the unfold/fold mechanism to handle recursive data structures. This obviates the need for specifying transitive closure relations that are used by classical verifier, such as Jahob, when tracking recursive properties. Lastly, as separation logic employs local reasoning via a frame rule, our approach does not require a separate modifies clause to be prescribed.

1.8 Conclusion

In summary, we have presented our software verification system HIP/SLEEK, which targets, but is not restricted to, C-like imperative programs. Assuming that each program is specified by means of pre-/post-conditions, our system verifies whether the program meets the specification, together with other memory safety and program liveness properties, such as safe pointer dereference, no dangling pointer, no memory leaks, and program termination. Even though not detailed in this chapter, HIP/SLEEK has also been enhanced for concurrency reasoning [48, 45, 46], safe barrier usage [47], for ensuring communication safety [19, 24], and also for reasoning about object-oriented programs [12]. Finally, to lift the users' burden of writing program specifications, HIP/SLEEK also makes available an inference system to analyze the code and to automatically synthesize its specifications.

References

1. https://github.com/hipsleek/hipsleek
2. http://loris-5.d2.comp.nus.edu.sg/TeachHIP/index.html
3. Barnett, M., Leino, R., Schulte, W.: The spec# programming system: An overview. In: CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices, pp. 49–69 (2005)

4. Berdine, J., Calcagno, C., O'Hearn, P.W.: A Decidable Fragment of Separation Logic. In: International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pp. 97–109 (2004)
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings, pp. 52–68 (2005)
6. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: memory safety for systems-level code. In: CAV'11, pp. 178–183. Springer-Verlag, Berlin, Heidelberg (2011). http://dl.acm.org/citation.cfm?id=2032305.2032320
7. Bozga, M., Iosif, R., Perarnau, S.: Quantitative Separation Logic and Programs with Lists. J. Autom. Reasoning **45**(2), 131–156 (2010)
8. Brotherston, J., Distefano, D., Petersen, R.L.: Automated Cyclic Entailment Proofs in Separation Logic. In: International Conference on Automated Deduction (CADE), pp. 131–146 (2011)
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300 (2009). http://doi.acm.org/10.1145/1480881.1480917
10. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: Symposium on Principles of Programming Languages (POPL), pp. 247–260 (2008)
11. Chen, C., Xi, H.: Combining programming with theorem proving. In: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005, pp. 66–77 (2005)
12. Chin, W., David, C., Nguyen, H.H., Qin, S.: Enhancing modular OO verification with separation logic. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, pp. 87–99 (2008)
13. Chin, W., Khoo, S.: Calculating sized types. In: Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22-23, 2000, pp. 62–72 (2000)
14. Chin, W., Khoo, S., Qin, S., Popeea, C., Nguyen, H.H.: Verifying safety policies with size properties and alias controls. In: 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, pp. 186–195 (2005)
15. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Multiple pre/post specifications for heap-manipulating methods. In: 10th IEEE High Assurance Systems Engineering Symposium (HASE'07), pp. 357–364. IEEE (2007)
16. Chu, D.H., Jaffar, J., Trinh, M.T.: Automatic induction proofs of data-structures in imperative programs. In: Conference on Programming Language Design and Implementation (PLDI), pp. 457–466 (2015)
17. Cok, D.R.: Reasoning with specifications containing method calls and model fields. Journal of Object Technology **4**(8), 77–103 (2005)
18. Cok, D.R., Kiniry, J.: Esc/java2: Uniting esc/java and JML. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers, pp. 108–128 (2004)
19. Costea, A., Chin, W., Qin, S., Craciun, F.: Automated modular verification for relaxed communication protocols. In: Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings, pp. 284–305 (2018)
20. Costea, A., Chin, W.N., Qin, S., Craciun, F.: Automated modular verification for relaxed communication protocols. In: S. Ryu (ed.) Programming Languages and Systems, pp. 284–305. Springer International Publishing, Cham (2018)
21. Costea, A., Sharma, A., David, C.: Hipimm: verifying granular immutability guarantees. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, pp. 189–193. ACM (2014)
22. Costea, M.A.: A session logic for relaxed communication protocols. Ph.D. thesis (2017)
23. Craciun, F., Kiss, T., Costea, A.: Towards a session logic for communication protocols. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 140–149 (2015). doi:https://doi.org/10.1109/ICECCS.2015.3310.1109/ICECCS.2015.33
24. Craciun, F., Kiss, T., Costea, A.: Towards a session logic for communication protocols. In: 20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia, December 9-12, 2015, pp. 140–149 (2015)

25. Curry, C., Le, Q.L., Qin, S.: Bi-Abductive Inference for Shape and Ordering Properties . In: 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 220–225. IEEE Computer Society, Los Alamitos, CA, USA (2019). https://doi.ieeecomputersociety.org/10.1109/ICECCS.2019.00031
26. David, C., Chin, W.N.: Immutable specifications for more concise and precise verification. ACM SIGPLAN Notices **46**(10), 359–374 (2011)
27. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
28. Elgaard, J., Klarlund, N., Møller, A.: MONA 1.x: New techniques for WS1S and WS2S. In: Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings, pp. 516–520 (1998). https://doi.org/10.1007/BFb0028773
29. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: Compositional Entailment Checking for a Fragment of Separation Logic. In: Asian Symposium on Programming Languages and Systems (APLAS), pp. 314–333 (2014)
30. Enea, C., Sighireanu, M., Wu, Z.: On automated lemma generation for separation logic with inductive definitions. In: International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 80–96 (2015)
31. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 2002, pp. 234–245 (2002)
32. Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, p. 927–942. Association for Computing Machinery, New York, NY, USA (2020). doi:https://doi.org/10.1145/3385412.338601410.1145/3385412.3386014
33. Gherghina, C., David, C., Qin, S., Chin, W.N.: Structured specifications for better verification of heap-manipulating programs. In: International Symposium on Formal Methods, pp. 386–401. Springer (2011)
34. Harwood, W., Cavalcanti, A., Woodcock, J.: A theory of pointers for the UTP. In: Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings, pp. 141–155 (2008)
35. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings, pp. 89–110 (1995)
36. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996, pp. 410–423 (1996)
37. Iosif, R., Rogalewicz, A., Simácek, J.: The Tree Width of Separation Logic with Recursive Definitions. In: International Conference on Automated Deduction (CADE), pp. 21–38 (2013)
38. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding Entailments in Inductive Separation Logic with Tree Automata. In: International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 201–218 (2014)
39. Ishtiaq, S.S., O'hearn, P.W.: Bi as an assertion language for mutable data structures. ACM SIGPLAN Notices **36**(3), 14–26 (2001)
40. Klarlund, N., Schwartzbach, M.I.: Graph types. In: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, pp. 196–205 (1993)
41. Kuncak, V., Lam, P., Zee, K., Rinard, M.C.: Modular pluggable analyses for data structure consistency. IEEE Trans. Software Eng. **32**(12), 988–1005 (2006)
42. Kuncak, V., Nguyen, H.H., Rinard, M.C.: An algorithm for deciding BAPA: boolean algebra with presburger arithmetic. In: Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings, pp. 260–277 (2005)

43. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006, pp. 115–126 (2006)
44. Lam, P.: The hob system for verifying software design properties. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2007)
45. Le, D., Chin, W., Teo, Y.M.: Variable permissions for concurrency verification. In: Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings, pp. 5–21 (2012)
46. Le, D., Chin, W., Teo, Y.M.: An expressive framework for verifying deadlock freedom. In: Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings, pp. 287–302 (2013)
47. Le, D., Chin, W., Teo, Y.M.: Verification of static and dynamic barrier synchronization using bounded permissions. In: Formal Methods and Software Engineering - 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1, 2013, Proceedings, pp. 231–248 (2013)
48. Le, D., Chin, W., Teo, Y.M.: Threads as resource for concurrency verification. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015, pp. 73–84 (2015)
49. Le, D.K., Chin, W.N., Teo, Y.M.: An expressive framework for verifying deadlock freedom. In: D. Van Hung, M. Ogawa (eds.) Automated Technology for Verification and Analysis, pp. 287–302. Springer International Publishing, Cham (2013)
50. Le, D.K., Chin, W.N., Teo, Y.M.: Verification of static and dynamic barrier synchronization using bounded permissions. In: L. Groves, J. Sun (eds.) Formal Methods and Software Engineering, pp. 231–248. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
51. Le, D.K., Chin, W.N., Teo, Y.M.: Threads as resource for concurrency verification. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM '15, p. 73–84. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2678015.2682540
52. Le, Q.L.: Compositional satisfiability solving in separation logic. In: F. Henglein, S. Shoham, Y. Vizel (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 578–602. Springer International Publishing, Cham (2021)
53. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape analysis via second-order bi-abduction. In: A. Biere, R. Bloem (eds.) Computer Aided Verification (CAV), pp. 52–68. Springer International Publishing, Cham (2014)
54. Le, Q.L., Le, X.B.D.: An efficient cyclic entailment procedure in a fragment of separation logic. In: Foundations of Software Science and Computation Structures: 26th International Conference, FoSSaCS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings, p. 477–497. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30829-1_23
55. Le, Q.L., Sun, J., Chin, W.N.: Satisfiability modulo heap-based programs. In: S. Chaudhuri, A. Farzan (eds.) Computer Aided Verification, pp. 382–404. Springer International Publishing, Cham (2016)
56. Le, Q.L., Sun, J., Qin, S.: Frame inference for inductive entailment proofs in separation logic. In: D. Beyer, M. Huisman (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 41–60 (2018)
57. Le, Q.L., Tatsuta, M., Sun, J., Chin, W.N.: A decidable fragment in separation logic with inductive predicates and arithmetic. In: R. Majumdar, V. Kunčak (eds.) Computer Aided Verification, pp. 495–517. Springer International Publishing, Cham (2017)
58. Le, T.C., Gherghina, C., Hobor, A., Chin, W.: A resource-based logic for termination and non-termination proofs. In: Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings, pp. 267–283 (2014). https://doi.org/10.1007/978-3-319-11737-9_18
59. Le, T.C., Qin, S., Chin, W.: Termination and non-termination specification inference. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pp. 489–498 (2015). https://doi.org/10.1145/2737924.2737993

60. Le, T.C., Ta, Q., Chin, W.: Hiptnt+: A termination and non-termination analyzer by second-order abduction - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II, pp. 370–374 (2017). https://doi.org/10.1007/978-3-662-54580-5_25
61. Le, X.B.D., Le, Q.L., Lo, D., Le Goues, C.: Enhancing automated program repair with deductive verification. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 428–432 (2016). doi:https://doi.org/10.1109/ICSME.2016.6610.1109/ICSME.2016.66
62. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes **31**(3), 1–38 (2006)
63. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers, pp. 348–370 (2010)
64. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL '10, POPL '10, pp. 211–222. ACM, New York, NY, USA (2010)
65. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001, pp. 221–231 (2001)
66. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, April 2008. Proceedings, pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
67. Nguyen, H.H., Chin, W.: Enhancing program verification with lemmas. In: Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings, pp. 355–369 (2008). https://doi.org/10.1007/978-3-540-70545-1_34
68. Nguyen, H.H., David, C., Qin, S., Chin, W.N.: Automated verification of shape and size properties via separation logic. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 251–266. Springer (2007)
69. Nguyen, T.T., Ta, Q.T., Chin, W.N.: Automatic program repair using formal verification and expression templates. In: C. Enea, R. Piskac (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 70–91. Springer International Publishing, Cham (2019)
70. Nguyen, T.T., Ta, Q.T., Sergey, I., Chin, W.N.: Automated repair of heap-manipulating programs using deductive synthesis. In: F. Henglein, S. Shoham, Y. Vizel (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 376–400. Springer International Publishing, Cham (2021)
71. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
72. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer (2002). doi:https://doi.org/10.1007/3-540-45949-910.1007/3-540-45949-9. https://doi.org/10.1007/3-540-45949-9
73. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using Separation Logic. In: Conference on Programming Language Design and Implementation (PLDI), p. 46 (2014)
74. Pérez, J.A.N., Rybalchenko, A.: Separation Logic + Superposition Calculus = Heap Theorem Prover. In: Conference on Programming Language Design and Implementation (PLDI), pp. 556–566 (2011)
75. Pérez, J.A.N., Rybalchenko, A.: Separation Logic Modulo Theories. In: Asian Symposium on Programming Languages and Systems (APLAS), pp. 90–106 (2013)
76. Peringer, P., Šoková, V., Vojnar, T.: PredatorHP Revamped (Not Only) for Interval-Sized Memory Regions and Memory Reallocation (Competition Contribution). In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 408–412. Springer International Publishing" (2020)
77. Pham, T.H., Trinh, M.T., Truong, A.H., Chin, W.N.: Fixbag: A fixpoint calculator for quantified bag constraints. In: CAV, pp. 656–662 (2011)
78. Piskac, R., Wies, T., Zufferey, D.: Automating Separation Logic Using SMT. In: International Conference on Computer Aided Verification (CAV), pp. 773–789 (2013)
79. Popeea, C., Chin, W.N.: Inferring disjunctive postconditions. In: ASIAN, pp. 331–345 (2006)
80. Popeea, C., Xu, D.N., Chin, W.N.: A practical and precise inference and specializer for array bound checks elimination. In: PEPM, pp. 177–187 (2008)

81. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991, pp. 4–13 (1991). https://doi.org/10.1145/125826.125848
82. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: Conference on Programming Language Design and Implementation (PLDI), pp. 231–242 (2013)
83. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE (2002)
84. Rugina, R.: Quantitative shape analysis. In: Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings, pp. 228–245 (2004)
85. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999, pp. 105–118 (1999)
86. Sharma, A., Hobor, A., Chin, W.N.: Specifying compatible sharing in data structures. In: M. Butler, S. Conchon, F. Zaïdi (eds.) Formal Methods and Software Engineering, pp. 349–365. Springer International Publishing, Cham (2015)
87. Sharma, A., Wang, S., Costea, A., Hobor, A., Chin, W.N.: Certified reasoning with infinity. In: International Symposium on Formal Methods, pp. 496–513. Springer (2015)
88. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated mutual explicit induction proof in separation logic. In: FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings, pp. 659–676 (2016)
89. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated Lemma Synthesis in Symbolic-Heap Separation Logic. In: Symposium on Principles of Programming Languages (POPL), pp. 9:1–9:29 (2018)
90. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated mutual induction proof in separation logic. Formal Aspect of Computing **31**(2), 207–230 (2019)
91. Trinh, M.T., Le, Q.L., David, C., Chin, W.N.: Bi-abduction with pure properties for specification inference. In: Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Australia. Proceedings, pp. 107–123 (2013). http://dx.doi.org/10.1007/978-3-319-03542-0_8
92. Tschannen, J.: Automatic verification of eiffel programs. Master's thesis, ETH Zurich (2007)
93. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999, pp. 214–227 (1999)