Go with the Flow: Compositional Abstractions for Concurrent Data Structures*

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

Concurrent separation logics have helped to significantly simplify correctness proofs for concurrent data structures. However, a recurring problem in such proofs is that data structure abstractions that work well in the sequential setting are much harder to reason about in a concurrent setting due to complex sharing and overlays. To solve this problem, we propose a novel approach to abstracting regions in the heap by encoding the data structure invariant into a local condition on each individual node. This condition may depend on a quantity associated with the node that is computed as a fixpoint over the entire heap graph. We refer to this quantity as a *flow*. Flows can encode both structural properties of the heap (e.g. the reachable nodes from the root form a tree) as well as data invariants (e.g. sortedness). We then introduce the notion of a *flow interface*, which expresses the relies and guarantees that a heap region imposes on its context to maintain the local flow invariant with respect to the global heap. Our main technical result is that this notion leads to a new semantic model of separation logic. In this model, flow interfaces provide a general abstraction mechanism for describing complex data structures. This abstraction mechanism admits proof rules that generalize over a wide variety of data structures. To demonstrate the versatility of our approach, we show how to extend the logic RGSep with flow interfaces. We have used this new logic to prove linearizability and memory safety of nontrivial concurrent data structures. In particular, we obtain parametric linearizability proofs for concurrent dictionary algorithms that abstract from the details of the underlying data structure representation. These proofs cannot be easily expressed using the abstraction mechanisms provided by existing separation logics.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Separation logic**; **Concurrent algorithms**; *Programming logic*;

Additional Key Words and Phrases: memory safety, linearizability, flow interfaces, separation algebra

ACM Reference Format:
Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2018. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *Proc. ACM Program. Lang.* 2, POPL, Article 37 (January 2018), 31 pages. https://doi.org/10.1145/3158125

1 INTRODUCTION

With the advent of concurrent separation logics (CSLs), we have witnessed substantial inroads into solving the difficult problem of concurrent data structure verification [Bornat et al. 2005; da Rocha Pinto et al. 2014, 2016; Dinsdale-Young et al. 2013, 2010; Dodds et al. 2016; Gu et al. 2015;

*This work is funded in parts by NYU WIRELESS and by the National Science Foundation under grants MCB-1158273, IOS-1339362, MCB-1412232, and CCF-1618059.

Authors' addresses: Siddharth Krishna, New York University, USA, siddharth@cs.nyu.edu; Dennis Shasha, New York University, USA, shasha@cims.nyu.edu; Thomas Wies, New York University, USA, wies@cs.nyu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
2475-1421/2018/1-ART37
https://doi.org/10.1145/3158125

Jung et al. 2015; Nanevski et al. 2014; O'Hearn 2004; Vafeiadis 2008; Vafeiadis and Parkinson 2007]. CSLs provide compositional proof rules that untangle the complex interference between concurrent threads operating on shared memory resources. At the heart of these logics lies separation logic (SL) [O'Hearn et al. 2001; Reynolds 2002]. The key ingredient of SL is the separating conjunction operator, which allows the global heap memory to be split into disjoint subheaps. An operation working on one of the subheaps can then be reasoned about compositionally in isolation using so-called frame rules that preserve the invariant of the rest of the heap. To support reasoning about complex data structures (lists, trees, etc.), SL is typically extended with predicates that are defined inductively in terms of separating conjunction to express invariants of unbounded heap regions. However, inductive predicates are not a panacea when reasoning about concurrent data structures.

One problem with inductive predicates is that the recursion scheme must follow a traversal of the data structure in the heap that visits every node exactly once. Such definitions are not well-suited for describing data structures with unbounded sharing and overlays of multiple data structures with separate roots whose invariants have mutual dependencies [Hobor and Villard 2013]. Both of these features are prevalent in concurrent algorithms (examples include B-link trees [Lehman and Yao 1981] and non-blocking lists with explicit memory management [Harris 2001]).

Another challenge is that proofs involving inductive predicates rely on lemmas that show how the predicates compose, decompose, and interact. For example, SL proofs of algorithms that manipulate linked lists often use the inductive predicate $\mathsf{lseg}(x, y)$, which denotes subheaps containing a list segment from x to y. A common lemma about lseg used in such proofs is that two disjoint list segments that share an end and a start point compose to a larger list segment (under certain side conditions). Unfortunately, these lemmas do not easily generalize from one data structure to another since the predicate definitions may follow different traversal patterns and generally depend on the data structure's implementation details. Hence, there is a vast literature describing techniques to derive such lemmas automatically, either by expressing the predicates in decidable fragments of SL [Berdine et al. 2004; Bouajjani et al. 2012; Cook et al. 2011; Enea et al. 2017; Iosif et al. 2014; Pérez and Rybalchenko 2011; Piskac et al. 2013; Reynolds et al. 2017; Tatsuta et al. 2016] or by using heuristics [Brotherston et al. 2011; Chlipala 2011; Enea et al. 2015; Nguyen and Chin 2008; Pek et al. 2014]. However, these techniques can be brittle, in particular, when the predicate definitions involve constraints on data.

What appears to be missing in existing SL variants is an abstraction mechanism for heap regions that is agnostic to specific traversal patterns, yet can still express inductive properties of the represented data structure. To address this shortcoming, we here take a radically different approach to abstraction in separation logic. Instead of relying on data-structure-specific inductive predicates, we introduce a new abstraction mechanism that specifies inductive structural and data properties uniformly but independently of each other, without fixing a specific traversal strategy.

As a first step, we describe data structure invariants in a uniform way by expressing them in terms of a condition local to each node of the heap graph. However, this condition is allowed to depend on a quantity associated with the node that is computed inductively over the entire graph. We refer to this inductively defined quantity as a *flow*. An example of a flow is the function pc that maps each node n to the number of paths from the root of the data structure to n. The property that a given heap graph is a tree can then be expressed by the condition that for all nodes n, $\mathrm{pc}(n) = 1$. Flows may also depend on the data stored in the heap graph. For instance, in a search data structure that implements a dictionary of key/value pairs, we can define the *inset flow*. The inset of a node n is a set of keys. Intuitively, a key k is in the inset of n if and only if a search for k that starts from the root of the data structure will have to traverse the node n. The inset flow can be used to express the invariants of search data structure algorithms in a way that abstracts from the concrete data

structure implementation (i.e., whether the concrete data structure is a list, a tree, or some more complicated structure with unbounded sharing) [Shasha and Goodman 1988].

We would like to reason compositionally about flows in SL using separating conjunctions. A separating conjunction is defined in terms of a composition operator on the semantic models of SL that yields a so-called *separation algebra* [Calcagno et al. 2007]. For the standard heap graph model of SL, composition is disjoint graph union: $G = G_1 \uplus G_2$. To enable local reasoning about flows, the composition operator needs to ensure that the flow on the resulting graph G maintains the local conditions with respect to the flows on its constituents G_1 and G_2. However, this is typically not the case for disjoint union. For instance, consider again our example of the path-counting flow where the flow condition bounds the path count of each node. Even if G_1 and G_2 individually satisfy the bound on the path count, $G_1 \uplus G_2$ may not since the union of the graphs can create cycles.

We therefore define a disjoint union operator on graphs that additionally asserts that the two subgraphs respect their mutual constraints on the flow of the composite graph. Our key technical contribution is to identify a class of flows that give rise to a separation algebra with this new composition operator. This class is closed under product (which enables reasoning about data structure overlays) and subsumes many natural examples of flows, including the ones given above.

To express abstract SL predicates that describe unbounded graph regions and their flows, we introduce the notion of a *flow interface*. A flow interface of a graph G expresses the constraints on G's contexts that G relies upon in order to satisfy its internal flow conditions, as well as the guarantees that G provides its contexts so that they can satisfy their flow conditions. In the example of path-counting flows used to express "treeness", the rely of the flow interface specifies how many paths G expects to exist from the global root in the composite graph to each of the roots in G (there can be many roots in G since it may consist of disconnected subtrees), and the guarantee specifies how many paths there are between each pair of root and sink node in G.

The algebraic properties of flows that guarantee that flow composition is well-defined also give rise to generic proof rules for reasoning about flow interfaces. These include rules that allow a flow interface to be split into arbitrary chunks which can be modified and recomposed, enabling reasoning about data structure algorithms that do not follow a fixed traversal strategy.

To demonstrate the usefulness of our new abstraction mechanism, we have instantiated rely/guarantee separation logic (RGSep) [Vafeiadis 2008; Vafeiadis and Parkinson 2007] with flow interfaces (other CSL flavors can be extended in a similar fashion). We have used the new logic to obtain simple correctness proofs of intricate concurrent data structure algorithms such as the Harris list with explicit memory management [Harris 2001], which cannot be easily verified with existing logics. Moreover, we show how the new logic can be used to formalize the edgeset framework for verifying concurrent dictionary data structures [Shasha and Goodman 1988]. We apply this formalization to proving linearizability [Herlihy and Wing 1990] of an algorithm template for concurrent dictionary implementations. The template abstracts from the specifics of the underlying data structure representation allowing it to be refined to diverse concrete implementations (e.g. using linked lists, B+ trees, etc.). By using flow interfaces, the correctness proof of the template can also abstract from the concrete implementation details, enabling proof reuse across the different refinements. We are not aware of any other logic that provides an abstraction mechanism to support this style of proof modularization at the level of data structure algorithms.

2 MOTIVATING EXAMPLE

Our aim is to prove memory safety and functional properties of concurrent data structures using local reasoning. We begin with a quick reminder of the standard way of specifying data structures in separation logic using inductive predicates. Using the example of the Harris list [Harris 2001], we see the limitations of the standard approach when dealing with concurrent data structures.

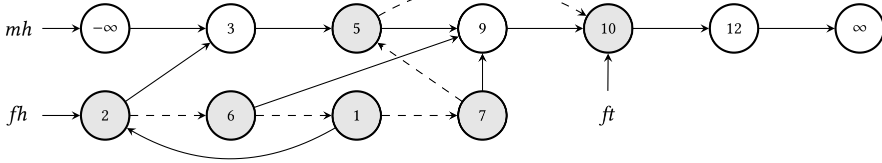

Fig. 1. A potential state of the Harris list with explicit memory management. `fnext` pointers are shown with dashed edges, marked nodes are shaded gray, and null pointers are omitted for clarity.

Separation logic (SL) is an extension of Hoare logic built for reasoning about programs that access and mutate data stored on the heap. Assertions in SL describe partial heaps (usually represented as a partial function from program locations to values), and can be interpreted as giving the program the permission to access the described heap region. Assertions are composed using the separating conjunction $P * Q$ which states that P and Q hold for disjoint regions of the heap. This enables local reasoning by means of a *frame rule* that allows one to remove regions of the heap that are not modified by a program fragment while reasoning about it.

The standard way of describing dynamic data structures (i.e. structures covering an unbounded and statically-unknown set of locations) in SL is using inductive predicates. For example, one can describe a singly-linked null-terminated acyclic list using the predicate

$$\mathsf{ls}(x) \coloneqq x = \mathit{null} \land \mathit{emp} \;\lor\; \exists y.\; x \mapsto y * \mathsf{ls}(y).$$

A heap h consists of a list from x if either $x = \mathit{null}$ and h is empty (emp) or h contains a location x whose value is y ($x \mapsto y$) and a *disjoint* null-terminated list beginning at y. The fact that the list is acyclic is enforced by the separating conjunction, which implies that x is not a node in $\mathsf{ls}(y)$.

One can write similar inductive predicates to describe any other data structure that can be unrolled and decomposed into subpredicates that do not share nodes. Sequential data structures such as sorted lists, doubly-linked lists, and binary search trees have been successfully verified in this manner. However, concurrent data structures often use intricate sharing and overlays.

For instance, consider the concurrent non-blocking linked list algorithm due to Harris [2001]. This algorithm implements a set data structure as a sorted list, and uses atomic compare-and-swap (CAS) operations to allow a high degree of parallelism. Harris' algorithm inserts a new key k into the list by finding nodes k_1, k_2 such that $k_1 < k < k_2$, setting k to point to k_2, and using a CAS to change k_1 to point to k only if it was still pointing to k_2. However, a similar approach fails for the delete operation. If we had consecutive nodes k_1, k_2, k_3 and we wanted to delete k_2 from the list (say by setting k_1 to point to k_3), there is no way to ensure with a single CAS that k_2 and k_3 are also still adjacent (another thread could have inserted or deleted in between them).

Harris' solution is a two step deletion: first atomically mark k_2 as deleted (by setting a mark bit on its successor field) and then later remove it from the list using a single CAS. After a node is marked, no thread can insert or delete to its right, hence a thread that wanted to insert k' to the right of k_2 would first remove k_2 from the list and then insert k' as the successor of k_1.

As described so far, this data structure can still be specified and verified using inductive predicates as the underlying shape is still a standard linked list. Things get complicated when we want to free deleted nodes. Since marked nodes can still be traversed by other threads, there may still be suspended threads accessing a marked node even after we remove it from the list. Thus, we cannot free the node immediately. A common solution is the so-called drain technique which maintains a second "free list" to which marked nodes are added before they are unlinked from the main list. These nodes are then labeled with a timestamp indicating when they were unlinked. When the

starting timestamp of all active threads is greater than the timestamp of a node in the free list, the node can be safely freed. This leads to the kind of data structure shown in Fig. 1, where each node has two pointer fields: a `next` field for the main list and an `fnext` field for the free list (shown as dashed edges). Threads that have been suspended while holding a reference to a node that was added to the free list can simply continue traversing the `next` pointers to find their way back to the unmarked nodes of the main list.

Even if our goal is to verify seemingly simple properties such as that the Harris list is memory safe and not leaking memory, the proof will rely on the following nontrivial invariants

(a) The data structure consists of two (potentially overlapping) lists: a list on `next` edges beginning at mh and one on `fnext` edges beginning at fh.
(b) The two lists are null terminated and ft is an element in the free list.
(c) The `next` edges from nodes in the free list point to nodes in the free list or main list.
(d) All nodes in the free list are marked.

One may be tempted to describe the Harris list simply as two lists (using the inductive predicate ls defined above), each using a different successor field. However, the two lists may overlap and so cannot be described by a separating conjunction of the two predicates. For the same reason, we can also not describe the entire structure with a single inductive predicate that uses separating conjunction because the predicate must describe each node exactly once[1]. Here, we could have arbitrarily many nodes in the free list with `next` edges to the same node in the main list (imagine a series of successive deletions of the predecessor of a node, such as 6, 7, 5 in Fig. 1). Because of this, we cannot describe the list nodes using separating conjunction as there is no way of telling if we are "double-counting" them. Also, using an inductive list predicate to describe the main list would work well only for reasoning about traversals starting from mh, since that is the only possible unrolling of the predicate. By contrast, threads may enter the main list at arbitrary points in the list by following `next` edges from the free list.

To further complicate matters, the `next` edges of free list nodes can even point backward to predecessors in the free list (for e.g. 1 points back to 2 in Fig. 1, creating a cycle). This means that even ramification-style approaches that use the overlapping conjunction [Hobor and Villard 2013] would not be able to specify this structure without switching to a coarse abstraction of an arbitrary potentially-cyclic graph (akin to indexed separating conjunctions [Reynolds 2002; Yang 2001]). However, a coarse graph abstraction could not easily capture key invariants such as that every node must be reachable from mh or fh, which are critical for proving the absence of memory leaks.

In the rest of this paper, we show how to solve these problems and obtain a compositional SL-based framework for reasoning about complex data structures like the Harris list. In §4 we define flows, show how they can be used to encode data structure invariants (both shape and data properties), build a flow-based semantic model for separation logic, and define an abstraction, flow interfaces, to reason compositionally about flow-based properties. §4.2 demonstrates the expressivity of our flow-based approach by showing how to describe many intricate data structures. We then extend a concurrent separation logic with flow interfaces in §5, show some lemmas to prove entailments in §5.5, and use them in §6 to verify memory safety and absence of memory leaks of a skeleton of the Harris list insert operation that abstracts from some of the algorithm's details but whose proof still relies on the above invariants (a) to (d). Our lemmas for reasoning about flows are data-structure-agnostic, i.e., they generalize across many different data structures. To demonstrate this feature, we show how to formalize the edgeset framework for concurrent dictionaries using flows in §7 and obtain a template algorithm with an implementation-agnostic proof of memory safety and linearizability.

[1]This is essentially because the disjointness of $*$ makes $x \mapsto _ * x \mapsto _$ unsatisfiable

3 PRELIMINARIES

Notation. The term $\mathsf{ITE}(b, t_1, t_2)$ denotes t_1 if condition b holds and t_2 otherwise. If f is a function from A to B, we write $f[x \rightarrowtail y]$ to denote the function from $A \cup \{x\}$ defined by $f[x \rightarrowtail y](z) :=$ $\mathsf{ITE}(z = x, y, f(z))$. We use $\{x_1 \rightarrowtail y_1, \ldots, x_n \rightarrowtail y_n\}$ for pairwise different x_i to denote the function $\epsilon[x_1 \rightarrowtail y_1] \cdots [x_n \rightarrowtail y_n]$, where ϵ is the function on an empty domain. If $f : A \times B \to C$, then we also write $\mathsf{dom}_1(f), \mathsf{dom}_2(f)$ for A, B respectively. For $f : A \to B$ and $C \subseteq A$ we write $f|_C : C \to B$ for the function obtained from f by restricting its domain to C. For $f_1 : A \to B$ and $f_2 : C \to B$ if $A \cap C = \emptyset$ then we write $f_1 \uplus f_2$ for the function $f : A \cup C \to B$ given by $f(x) := \mathsf{ITE}(x \in A, f_1(x), f_2(x))$. Moreover, we denote by $f(C)$ the set $\{f(c) \mid c \in C\}$.

Semirings, ω-cpos, and continuous functions. A semiring $(D, +, \cdot, 0, 1)$ is a set D equipped with binary operators $+, \cdot : D \times D \to D$. The operation $+$ is called addition, and the operation $\cdot$ multiplication. The two operators must satisfy the following properties: (1) $(D, +, 0)$ is a commutative monoid with identity 0; (2) $(D, \cdot, 1)$ is a monoid with identity 1; (3) multiplication left and right distributes over addition; and (4) multiplication with 0 annihilates D, i.e., $0 \cdot d = d \cdot 0 = 0$ for all $d \in D$.

An ω-complete partial order (ω-cpo) is a set D equipped with a partial order $\sqsubseteq$ on D such that all increasing chains in D have suprema in D. A function $f : D_1 \to D_2$ between two ω-cpos $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ is continuous if $f(\bigsqcup_1 X) = \bigsqcup_2 f(X)$ for any increasing chain $X \subseteq D_1$. Here, $\bigsqcup_i X$ denotes the supremum of a chain X in D_i.

4 THE FLOW FRAMEWORK

This section presents the formal treatment of our flow framework.

Our theory is parameterized by the domain over which flows range. This domain is equipped with operations for calculating flows, which must satisfy certain algebraic properties.

Definition 4.1 (Flow Domain). A *flow domain* $(D, \sqsubseteq, \sqcup, +, \cdot, 0, 1)$ is a positive partially ordered semiring that is ω-complete. That is, (1) $(D, \sqsubseteq)$ is an ω-cpo and $\sqcup$ its join; (2) $(D, +, \cdot, 0, 1)$ is a semiring; (3) $+$ and $\cdot$ are continuous with respect to $\sqsubseteq$; and (4) 0 is the smallest element of D. We identify a flow domain with its support set D.

Example 4.2. The natural numbers extended with infinity, $\mathbb{N}^\infty := (\mathbb{N} \cup \{\infty\}, \leq, \max, +, \cdot, 0, 1)$, form a flow domain and so does $(D, \sqsubseteq, \sqcup, \sqcup, \sqcap, \bot, \top)$ for any completely distributive lattice $(D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.

In our formalization, we consider graphs whose nodes are labeled from some set of node labels A that encode relevant information contained in each node (e.g. the node's content or the id of the thread holding a lock on the node). We will later build abstractions of graphs that compute summaries of these node labels. For this purpose, we require that A is equipped with a partial order $\sqsubseteq$ that induces a join-semilattice $(A, \sqsubseteq, \sqcup, a_e)$ with join $\sqcup$ and smallest element a_e.

For the remainder of this section we fix a flow domain $(D, \sqsubseteq, \sqcup, +, \cdot, 0, 1)$ and node label domain $(A, \sqsubseteq, \sqcup, a_e)$. We use the same symbols for the partial order and join operator on the two domains. However, it will always be clear from the context which one is meant. All definitions are implicitly parameterized by D and A.

4.1 Flows

Flows are calculated over directed graphs that serve as an intermediate abstraction of heap graphs. The graphs are partial, i.e., they may have outgoing edges to nodes that are not itself part of the graph. The edges are labeled by elements of the flow domain and the nodes with node labels. Formally, given a (potentially infinite) set of nodes $\mathcal{N}$, a *(partial) graph* $G = (N, N^o, \lambda, \varepsilon)$ consists

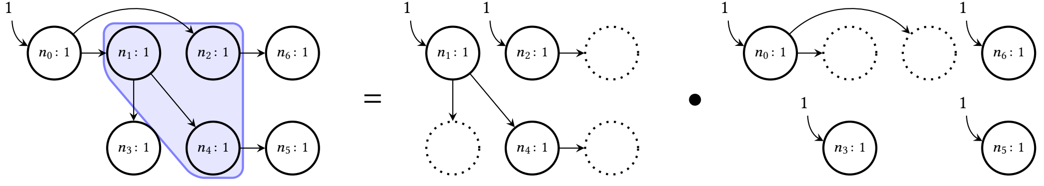

Fig. 2. A decomposition of an inflow/graph pair (in, G) into the boxed blue region (in_1, G_1) and its context (in_2, G_2). All shown edges have edge label 1, missing edges have label 0, and the path-counting flow is used. Non-zero inflows are shown as curved arrows at each node, nodes are labeled with names and the resulting flows, and sink nodes are shown as dotted circles.

of a finite set of nodes $N \subseteq \mathcal{N}$, a finite set of *sink nodes* $N^o \subseteq \mathcal{N}$ disjoint from N, a node labeling function $\lambda\colon N \to A$, and an edge function $\varepsilon\colon N \times (N \cup N^o) \to D$. Note that the edge function is total on $N \times N^o$. The absence of an edge between two nodes n, n' is indicated by $\varepsilon(n, n') = 0$. We let $\mathrm{dom}(G) = N$ and sometimes identify G and $\mathrm{dom}(G)$ to ease notational burden. The (unique) graph defined over the empty set of nodes and sinks is denoted by G_e. We define the disjoint union of two graphs $G_1 \uplus G_2$ in the expected way:

$$G_1 \uplus G_2 \coloneqq \begin{cases} (N_1 \cup N_2, (N_1^o \setminus N_2) \cup (N_2^o \setminus N_1), \lambda_1 \uplus \lambda_2, \varepsilon') & \text{if } N_1 \cap N_2 = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{where } \varepsilon'(n_1, n_2) \coloneqq \begin{cases} \varepsilon_1(n_1, n_2) & n_1 \in N_1 \wedge n_2 \in N_1 \cup N_1^o \\ \varepsilon_2(n_1, n_2) & n_1 \in N_2 \wedge n_2 \in N_2 \cup N_2^o \\ 0 & \text{otherwise.} \end{cases}$$

A graph G_1 is a subgraph of another graph G, denoted $G_1 \subseteq G$, if and only if there exists G_2 such that $G = G_1 \uplus G_2$.

Example 4.3. The graph G_1 in the middle of Fig. 2 is $(\{n_1, n_2, n_4\}, \{n_3, n_5, n_6\}, \lambda_1, \varepsilon_1)$ for some λ_1 and an ε_1 that sets all pairs in $\{(n_1, n_3), (n_1, n_4), (n_2, n_6)\}$ to 1 and all other pairs to 0. Similarly, G_2 on the right is $(\{n_0, n_3, n_5, n_6\}, \{n_1, n_2\}, \lambda_2, \varepsilon_2)$ for appropriate λ_2 and ε_2, and the composed graph $G = G_1 \uplus G_2$ is shown on the left of Fig. 2.

Let $G = (N, N^o, \lambda, \varepsilon)$ be a graph. A *flow* of G is a function $\mathrm{flow}(in, G)\colon N \to D$ that is calculated as a certain fixpoint over G's edge function starting from a given *inflow* $in\colon N \to D$ going into G. To motivate the definition of $\mathrm{flow}(in, G)$, let us use the flow domain $\mathbb{N}^\infty$ from Example 4.2. We can use this flow domain to define a flow that counts the number of paths between a fixed node and any other node. First, view the edge function ε as an adjacency matrix E indexed by the nodes $N \cup N^o$. If one looks at the matrix product $E \cdot E = E^2$ (using the addition and multiplication operations of the flow domain), then the (n, n')-th entry corresponds to the sum over the products of edge labels on all paths of length 2 between n and n'. For G from Fig. 2, for example, the (n_0, n_3)-th entry of E^2 is 1 as there is exactly 1 non-zero length 2 path, while the (n_0, n_1)-th entry is 0 as there is no non-zero length 2 path. Extending this idea, for any graph where edges are labeled from $\{0, 1\}$, the (n, n')-th entry of matrix E^l is the number of l length paths from n to n'. Thus, the matrix $C = I + E + E^2 + \dots$ tells us the number of all paths between two nodes. Given a root node r, we can look at the r-th row of this matrix to get the number of paths to any node from the root. To generalize this to multiple roots, we can calculate the number of paths to any node by looking at $\vec{in} \cdot C$ where $\vec{in}$ is a row vector with a 1 for every root node and 0 otherwise.

Formally, first let the *capacity* $\mathsf{cap}(G)\colon N \times (N \cup N^o) \to D$ be the least fixpoint of the following equation:

$$\mathsf{cap}(G)(n, n') = \mathsf{init}(n, n') + \sum_{n'' \in G} \varepsilon(n, n'') \cdot \mathsf{cap}(G)(n'', n') \quad \text{where} \quad \mathsf{init}(n, n') \coloneqq \mathsf{ITE}(n = n', 1, 0).$$

Note that the fixpoint exists because D is ω-complete and $+$ and $\cdot$ are continuous. Then define

$$\mathsf{flow}(in, G)(n) \coloneqq \sum_{n' \in G} in(n') \cdot \mathsf{cap}(G)(n', n).$$

Example 4.4. In Fig. 2, the capacity of G_2 is equal to its edge function, the capacity of G_1 is its edge function with (n_1, n_5) additionally 1, and $\mathsf{cap}(G)$ is 1 for all pairs (n, n') such that n is an ancestor of n' in the tree. For the inflow $in(x) = \mathsf{ITE}(x = n_0, 1, 0)$, $\mathsf{flow}(in, G) = (\lambda x.1)$.

Finally, a key property of flows is that they can be composed by considering the product flow on the product of their flow domains.

Lemma 4.5. *Let* $(D_1, \sqsubseteq_1, \sqcup_1, +_1, \cdot_1, 0_1, 1_1)$ *and* $(D_2, \sqsubseteq_2, \sqcup_2, +_2, \cdot_2, 0_2, 1_2)$ *be flow domains.* $(D_1 \times D_2, \sqsubseteq, \sqcup, +, \cdot, (0_1, 0_2), (1_1, 1_2))$*, where* $\sqsubseteq, \sqcup, +$*, and* $\cdot$ *operate on each component respectively, is a flow domain. Moreover, the flow on the product domain is the product of the flows on each component.*

4.2 Expressivity

We now give a few examples of flows to demonstrate the range of data structures that can be described using local conditions on flows.

The shape of all common structures that can be described with inductive predicates can be described using flows, e.g. lists (singly and doubly linked, cyclic), trees, and nested combinations of these. By considering products with flows for data properties, we can also describe structures such as sorted lists, binary heaps, and search trees. Going beyond inductive predicates, the flow framework can describe structures with unbounded sharing and irregular traversals (e.g. Harris list), overlays (e.g. threaded and B-link trees), as well as those with irregular shape (e.g. DAGs and arbitrary graphs).

In this section, we assume that the local condition on the flow of each node is specified by a *good condition.* This is a predicate $\mathsf{good}(n, in, a, f)$ taking as arguments the node n, the inflow function in on the singleton graph containing n, the node label a of n, and the edge function f of that singleton graph. The singleton inflow of n is equal to the flow at n in the global graph representing the data structure of interest containing n. This fact, as well as the reason for this formalization, will be seen in §4.4. We also assume that we can specify the global inflow of our data structure.

4.2.1 Path-counting Flow. Consider the flow domain $\mathbb{N}^\infty$ from Example 4.2, with an arbitrary node label domain.

Lemma 4.6. *For any graph G, if the edge label of every edge* (n, n') *is interpreted as the number of edges between n and n′, then* $\mathsf{cap}(G)(n, n')$ *is the number of paths from n to n′. If the inflow in takes values in* $\{0, 1\}$ *then the path-counting flow* $\mathsf{flow}(in, G)(n)$ *is the number of paths from nodes in* $\{n' \mid in(n') = 1\}$ *to n.*

We can use this flow to describe shapes such as lists and trees. For instance, a graph is a tree if and only if every node has exactly one path from the root. Thus, if a graph G satisfies the good condition

$$\mathsf{good}_{\mathsf{tr}}(n, in, _, _) \coloneqq in(n) = 1$$

at all nodes n, given a global inflow satisfying $in(n) = \mathsf{ITE}(n = n_0, 1, 0)$, then G must be a tree rooted at n_0. As a singly-linked list is a tree where every node has exactly one outgoing edge, we can

describe it using the good condition

$$\mathsf{good}_{\mathsf{ls}}(n, in, _, f) := \exists n'.\ in(n) = 1 \wedge (f = \epsilon \vee f = \{(n, n') \rightarrowtail 1\})$$

where ϵ stands for the empty function (i.e. the node has no outgoing edges). A graph G satisfying $\mathsf{good}_{\mathsf{ls}}$ under a global inflow in with $in(n) = \mathsf{ITE}(n = n_0, 1, 0)$ for all n must be a list beginning at n_0. To model a null-terminated list, we must further require that the entire graph G has no outgoing edges. This can be done using our flow interface abstraction in §4.4. If we want to model a list that terminates at a specific node, say ft as in the free list shown in Fig. 1, we can replace the last clause of $\mathsf{good}_{\mathsf{ls}}$ with $f = \mathsf{ITE}(n = ft, \epsilon, \{(n, n') \rightarrowtail 1\})$.

Similarly, one can describe cyclic lists by requiring that each node has exactly one outgoing edge and that the path count of each node is ∞. To describe a tree of a particular arity, like a binary tree, one can constrain the path count to be 1 and add a further condition restricting the number of outgoing edges at each node appropriately. For overlaid structures like the threaded tree, which is a tree where all elements also form a list (in an arbitrary order), we can use the product of two path-counting flows, one from the root of the tree and one from the head of the list.

4.2.2 Last-edge Flow. We next show a flow domain where the flow along a path is equal to the label of the last edge on the path.

LEMMA 4.7. *Given an ω-cpo $(D, \sqsubseteq, \sqcup)$ with smallest element 0, let 1 be a fresh element. There exists a flow domain $(D \uplus \{1\}, \sqsubseteq, \sqcup, \sqcup, \rtimes, 0, 1)$, where multiplication is the projection operator, defined as $d_1 \rtimes d_2 := \mathsf{ITE}(d_1 = 0, 0, \mathsf{ITE}(d_2 = 1, d_1, d_2))$ and the ordering is extended with $0 \sqsubseteq 1$.*

This flow has many applications; for instance, we can start with a lattice of types and use the last-edge flow to enforce that all reachable nodes are well-typed in the presence of type-unsafe operations. Another example is doubly linked lists, which can be described by using a product of two path-counting flows, one counting paths from the head and the other from the tail, and a last-edge flow to ensure that for any node `n`, `n.next.prev` is equal to `n`.

We can also use this flow to encode nested combinations of structures such as trees of lists. We start with a path-counting flow to enforce that the entire graph forms a tree and then take a product with the last-edge flow domain built from the ω-cpo on $\{0, d_T, d_L\}$ where 0 is the smallest element and the others are unordered. We then use the good condition to label all edges to tree nodes with d_T, all edges to list nodes with d_L, and enforce that the label on the incoming edge matches the type of the node and that list nodes have no edges to tree nodes.

4.2.3 Upper and Lower-bound Flows. We next turn to data properties like sortedness. Consider the lower-bound flow domain formed by the integers extended with positive and negative infinities: $(\mathbb{Z} \cup \{-\infty, \infty\}, \geq, \min, \min, \max, \infty, -\infty)$. Assume further that the node label domain consists of sets of data values and that each node is labeled with the singleton set of its content. If we label each edge of a list with the value at the source node and start with an inflow of $-\infty$ at the root h, then the flow at a node is the maximum value of its predecessors. This value is like a lower bound for the value of the current node; if the value at every node reachable from h is greater than its lower-bound flow, then the list beginning at h is sorted in ascending order. To do this, we use the product of the path-counting and the lower-bound flow domains. We can use the good condition

$$\mathsf{good}_{\mathsf{sls}}(n, in, a, f) := \exists n', l, k.\ in(n) = (1, l) \wedge a = \{k\} \wedge f = \mathsf{ITE}(n' = null, \epsilon, \{(n, n') \rightarrowtail k\}) \wedge l \leq k$$

and a global inflow $in(n) = \mathsf{ITE}(n = n_0, (1, -\infty), (0, \infty))$ to describe a sorted list beginning at n_0.

Note that, unlike in the definitions of inductive SL predicates, the data constraints are not tied to the shape constraints – we could just as easily use the lower-bound flow condition of sortedness and the path-counting flow condition of being a tree to describe a min-heap.

Analogously, we can use the upper-bound flow domain $(\mathbb{Z} \cup \{-\infty, \infty\}, \leq, \max, \max, \min, -\infty, \infty)$ to describe lists sorted in descending order and max-heaps. We can also use a product of a lower-bound and an upper-bound flow to describe a binary search tree by enforcing that each node propagates to its child the appropriate bounds on values that can be present in the child's subtree.

We will see a more general flow, the inset flow, that can be used to specify data properties of dictionary data structures that subsumes the examples above in §7.

4.3 Flow Graph Algebras

We next show how to define a separation algebra that can be used to give semantics to separation logic assertions in a way that allows us to maintain flow-dependent invariants. This will enable us to reason locally about graphs and their flows using the $*$ operator of SL.

Definition 4.8 (Separation Algebra [Calcagno et al. 2007]). A *separation algebra* is a cancellative, partial commutative monoid $(\Sigma, \bullet, u)$. That is, $\bullet \colon \Sigma \times \Sigma \rightharpoonup \Sigma$ and $u \in \Sigma$ such that the following properties hold:

(1) Identity: $\forall \sigma \in \Sigma.\ \sigma \bullet u = \sigma$.
(2) Commutativity: $\forall \sigma_1, \sigma_2 \in \Sigma.\ \sigma_1 \bullet \sigma_2 = \sigma_2 \bullet \sigma_1$.
(3) Associativity: $\forall \sigma_1, \sigma_2, \sigma_3 \in \Sigma.\ \sigma_1 \bullet (\sigma_2 \bullet \sigma_3) = (\sigma_1 \bullet \sigma_2) \bullet \sigma_3$.
(4) Cancellativity: $\forall \sigma, \sigma_1, \sigma_1', \sigma_2 \in \Sigma.\ \sigma = \sigma_1 \bullet \sigma_2 \land \sigma = \sigma_1' \bullet \sigma_2 \Rightarrow \sigma_1 = \sigma_1'$.

Here, equality means that either both sides are defined and equal, or both sides are undefined.

A simple way of obtaining a separation algebra of graphs is to define the composition $\bullet$ in terms of disjoint union. Effectively, this yields the standard model of separation logic. However, suppose we have a graph G satisfying some local condition on the flow at each node that we want to decompose into two subgraphs G_1 and G_2, say to verify a program that traverses the Harris list. To infer that the subgraph G_1 also satisfies the same local condition on the flow, we need to be able to compute the flow of nodes in G_1 without looking at the entire graph. We thus want a stricter version of composition that allows us to decompose a flow on G into flows on G_1 and G_2. Our solution is to use graphs equipped with inflows as the elements of our separation algebra.

Since we allow our graphs to have cycles, it is hard to define the composed inflow of two inflows on subgraphs. Instead, we start with an inflow/graph pair (in, G) and show how to decompose it into an inflow for its constituent subgraphs G_i. To do this, we *project* the inflow in to obtain a new inflow function in_i for G_i that results in the same flow at each node of G_i as in on G. The projected inflow $in_1(n)$ of a node in G_1 is equal to $in(n)$ plus the contribution of G_2 to the flow of n (and vice versa for in_2 on G_2). We will use projection to define the composition of inflow/graph pairs.

Definition 4.9. The projection of an inflow in on a graph $G = G_1 \uplus G_2$ onto G_1 is a function $\mathsf{proj}(in, G)(G_1) \colon \mathsf{dom}(G_1) \to D$ defined as:

$$\mathsf{proj}(in, G)(G_1)(n) \coloneqq in(n) + \sum_{n' \in G \setminus G_1} \mathsf{flow}(in, G)(n') \cdot \varepsilon(n', n).$$

The following lemma tells us that the flow induced by the projection of an inflow in onto a subgraph is the same as the flow induced by in on the larger graph. In other words, we can view the inflow as containing all information about the context of a graph that is needed for calculation of the flow of nodes in the graph.

LEMMA 4.10. *If in is an inflow on G and $in_1 = \mathsf{proj}(in, G)(G_1)$ for some $G_1 \subseteq G$, then* $\mathsf{flow}(in, G)|_{G_1} = \mathsf{flow}(in_1, G_1)$.

Example 4.11. Fig. 2 uses the path-counting flow and shows a split of a graph G into the boxed blue region G_1 and its context G_2. The inflow on G is $in(n) = \mathsf{ITE}(n = n_0, 1, 0)$, and its projections onto G_1 and G_2 are shown as curved arrows entering the nodes in each graph.

We next define a first approximation of a composition operator on inflow/graph pairs that preserves the flow at each node. The composition of two inflow/graph pairs (in_1, G_1) and (in_2, G_2), denoted $(in_1, G_1) \bullet (in_2, G_2)$, is the set of all pairs (in, G) such that G_1 and G_2 compose to G and the component inflows in_1 and in_2 are the projections of in onto the respective subgraphs:

$$(in, G) \in (in_1, G_1) \bullet (in_2, G_2) :\Leftrightarrow G = G_1 \uplus G_2 \wedge \forall i \in \{1, 2\} \,.\, \mathsf{proj}(in, G)(G_i) = in_i .$$

Lemma 4.10 then ensures that the flow is the same in the composite graph as in the subgraphs. Note that this set may be empty even if G_1 and G_2 are disjoint since the inflows of the two subgraphs may not be compatible.

Unfortunately, $(in_1, G_1) \bullet (in_2, G_2)$ may contain more than one element since there may be many possible composite inflows in whose projection onto the subgraphs yields in_1 and in_2. For example, let G_1, G_2 be singleton graphs on nodes n_1, n_2 respectively that each have an edge with label $\infty \in \mathbb{N}^\infty$ to the other. If $in_1 = \{n_1 \rightarrowtail \infty\}$ and $in_2 = \{n_2 \rightarrowtail \infty\}$, then the composite inflow can be $\{n_1 \rightarrowtail 1\}$ or $\{n_2 \rightarrowtail 1\}$ (among many others). So we cannot immediately use $\bullet$ to define the partial monoid for our separation algebra. However, as we shall see, all inflows in satisfying the condition of the composition induce the same flow on G. Thus, let us define an equivalence relation $\sim_G$ on inflows for a given graph G. The relation $\sim_G$ relates all inflows that induce the same flows:

$$in \sim_G in' :\Leftrightarrow \mathsf{flow}(in, G) = \mathsf{flow}(in', G).$$

Given an inflow in, we denote its equivalence class with respect to $\sim_G$ as $[in]_G$ and extend flow from inflows to their equivalence classes in the expected way. We also write $\mathsf{dom}(In) := \mathsf{dom}(in)$ for $In = [in]_G$, and use IN to denote the set of all inflow equivalence classes.

Using Lemma 4.10 we can show that the equivalence relation $\sim$ is a congruence on $\bullet$:

Lemma 4.12. *The following two implications hold:*

(1) $(in, G) \in (in_1, G_1) \bullet (in_2, G_2) \wedge in_1 \sim_{G_1} in_1' \wedge in_2 \sim_{G_2} in_2' \wedge (in', G) \in (in_1', G_1) \bullet (in_2', G_2) \Rightarrow in \sim_G in'$.

(2) $(in, G) \in (in_1, G_1) \bullet (in_2, G_2) \wedge in \sim_G in' \Rightarrow$ $\exists in_1', in_2' .\ in_1 \sim_{G_1} in_1' \wedge in_2 \sim_{G_2} in_2' \wedge (in', G) \in (in_1', G_1) \bullet (in_2', G_2)$.

Lemma 4.12 implies that $\bullet$ yields a partial function on pairs of graphs G and their inflow equivalence classes. This suggests the following definition for our separation algebra.

Definition 4.13 (Flow Graph Algebra). The *flow graph algebra* $(\mathsf{FG}, \bullet, H_e)$ for flow domain D and node label domain A is defined by

$$H \in \mathsf{FG} := \{([in]_G, G) \mid in \colon \mathsf{dom}(G) \to D\}$$

$$([in_1]_{G_1}, G_1) \bullet ([in_2]_{G_2}, G_2) := \begin{cases} ([in]_G, G) & G = G_1 \uplus G_2 \wedge \forall i \in \{1, 2\} \,.\, \mathsf{proj}(in, G)(G_i) \in [in_i]_{G_i} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$H_e := ([in_e]_{G_e}, G_e)$$

where G_e is the empty graph and in_e the inflow on an empty domain. We call the elements $H \in \mathsf{FG}$ *flow graphs*. We again let $\mathsf{dom}(H) = \mathsf{dom}(G)$ and write H for $\mathsf{dom}(H)$ when it is clear from the context.

Theorem 4.14. *The flow graph algebra* $(\mathsf{FG}, \bullet, H_e)$ *is a separation algebra.*

4.4 Abstracting Flow Graphs with Flow Interfaces

We can now use flow graphs to give semantics to separation logic assertions. However, we also want to be able to use predicates in the logic that describe sets of such graphs so that the flow of each graph in the set satisfies certain local invariants at each of its nodes. In the following, we introduce semantic objects, which we call *flow interfaces*, that provide such abstractions. We will then lift the composition operator $\bullet$ from flow graphs to flow interfaces and prove important properties about flow interface composition. In §5, we will use these properties to justify general lemmas for proving entailments in a separation logic with flow interfaces.

Our target abstraction must have certain properties. Recall from §4.2 that to describe data structure invariants as conditions on the flow we fixed a particular inflow to the global graph. Thus, our abstraction must fix the inflow of the abstracted graph. Secondly, our abstraction must be sound with respect to flow graph composition: if we have a graph $G = G_1 \uplus G_2$, and G_1 and G_1' satisfy the same interface, then $G_1' \uplus G_2$ must be defined and satisfy the same interface as G.

The second property is a challenging one, for the effect of a local modification in a flow graph are not necessarily local: small changes in G_1 may affect the flow of nodes in G_2. For example, in Fig. 2 if we modified G_1 by adding a new outgoing edge from n_1 to n_6, then this would increase the path count of n_6 to 2 and we can no longer compose with G_2 as it expects an inflow of 1 at n_6. We need to find a characterization of modifications to G_1 that preserve the inflow of every node in G_2.

Intuitively, the internal structure of G_1 should be irrelevant for computing the flow of G_2; from the perspective of G_2, G_1 should be replaceable by any other flow graph that provides G_2 with the same inflow. One may try to define an outflow quantity, analogous to inflow, that specifies the flow that a graph gives to each of its sink nodes. If the interface abstracting G_1 also fixes its outflow, then one expects the flow of G_2 to be preserved. However, this is not true in the presence of cycles.

On the other hand, if we preserve the capacity between every pair of node and sink node in the modification of G_1 to G_1', then the flow of all nodes in G_2 stays the same. This is a consequence of the semiring properties of the flow domain and the fact that $+$ and $\cdot$ are continuous. Moreover, as G_2 can only route flow into nodes in G_1 with a non-zero inflow, we only need to preserve the capacity from these *source* nodes. We capture this idea formally by defining the *flow map* of $H = (In, G)$ as the restriction of $\mathsf{cap}(G)$ onto G's source-sink pairs as specified by In:

$$\mathsf{flm}((In, G)) \coloneqq \mathsf{cap}(G)|_{\{n \in N \mid \exists in \in In.\ in(n) > 0\} \times N^o}.$$

This gives us sufficient technical machinery to define our abstraction of flow graphs.

Definition 4.15 (Flow Interface). Given a flow graph $(In, G) \in \mathsf{FG}$, its *flow interface* is the tuple consisting of its inflow equivalence class, the join of all its node labels, and its flow map:

$$\mathsf{int}(H) \coloneqq (In, \sqcup_{n \in H} \lambda(n), \mathsf{flm}(H)) \quad \text{where } H = (In, G) \text{ and } G = (N, N^o, \lambda, \varepsilon).$$

The set of all flow interfaces is $\mathsf{FI} \coloneqq \{\mathsf{int}(H) \mid H \in \mathsf{FG}\}$. The denotation of a flow interface is a set of flow graphs defined as $[\![I]\!] \coloneqq \{H \in \mathsf{FG} \mid \mathsf{int}(H) = I\}$.

Example 4.16. The interface of $H_1 = ([in_1]_{G_1}, G_1)$ from Fig. 2 is $I_1 = (\{in_1\}, a_e, f_1)$ where f_1 is the flow map that sets $\{(n_1, n_3), (n_1, n_5), (n_2, n_6)\}$ to 1 and every other pair to 0 (assuming all nodes are labeled with a_e). If we now modified H_1 to H_1' by removing the edge (n_1, n_4) and adding an edge (n_1, n_5), then H_1' will also be in the denotation $[\![I_1]\!]$. Note that H_1' still composes with the flow graph on G_2, and the flow at all nodes in G_2 is the same in this composition.

If we look at a flow interface $I = (In, a, f)$ in the rely-guarantee setting, a flow graph satisfying I relies on getting some inflow in In from its context, and guarantees the flow map f to its context. The third component, a, provides a summary of the information contained in the nodes of a graph,

which will be useful, for example, to reason about the contents of data structures. We identify the domain of an interface with the domain of the graphs it abstracts, i.e. $\mathsf{dom}((In, a, f)) = \mathsf{dom}(In)$.

Note that both the inflow class and flow map of a flow interface I fully determine the domains of the graphs being described by the interface. That is, I does not abstract from the identity of the nodes in the graphs $[\![I]\!]$, which is intentional as it allows us to define separation logic predicates in terms of flow interfaces that have precise semantics, without encoding specific traversal patterns of the underlying data structure and without restricting sharing as other methods do.

If we consider the equivalence relation on flow graphs induced by having the same flow interface, then we have the required property that this relation is a congruence on $\bullet$.

Lemma 4.17. *If $H_1, H_1' \in [\![I_1]\!]$, $H_2, H_2' \in [\![I_2]\!]$ and $H_1 \bullet H_2 \in [\![I]\!]$, then $H_1' \bullet H_2' \in [\![I]\!]$.*

We can now lift flow graph composition to flow interfaces, denoted $I = I_1 \oplus I_2$, as follows:

$$I = I_1 \oplus I_2 :\Leftrightarrow \exists H \in [\![I]\!], H_1 \in [\![I_1]\!], H_2 \in [\![I_2]\!].\ H = H_1 \bullet H_2.$$

Note that $\oplus$ is also a partial function for the domains of the graphs may overlap, or the inflows may not be compatible. The identity flow interface I_e is defined by $I_e := (\{in_\mathrm{e}\}, a_e, f_\mathrm{e})$, where f_e is the flow map on an empty domain. A simple corollary of Lemma 4.17 is that the flow interface algebra $(\mathsf{FI}, \oplus, I_e)$ is also a separation algebra.

As we saw in §4.2, to express properties of different data structures we need to use abstractions of flow graphs that additionally satisfy certain conditions on the flow at each node. To formalize this, we assume the node condition is specified by a predicate $\mathrm{good}(n, in, a, f)$ which takes a node n, the inflow in of the singleton flow graph containing n, the node label a of n, and the edge function f of that singleton graph as arguments. The denotation of a flow interface I with respect to a good condition good is defined as

$$[\![I]\!]_{\mathrm{good}} := \{(In, (N, N^o, \lambda, \varepsilon)) \in [\![I]\!] \mid \forall n \in N.\ \mathrm{good}(n, in_n, \lambda(n), \varepsilon_n)\}$$

where $in_n := \{n \rightarrowtail \mathsf{flow}(In, G)(n)\}$ and $\varepsilon_n := \{(n, n') \rightarrowtail \varepsilon(n, n') \mid n' \in N \cup N^o, \varepsilon(n, n') \neq 0\}$. Note that the interface of the singleton flow graph containing n is $(\{in_n\}, \lambda(n), \mathsf{ITE}(in_n(n) = 0, \epsilon, \varepsilon_n))$, but we choose the above formulation for notational convenience.

The following lemma tells us that imposing a good condition on flow interfaces is essentially reducing the domain of flow graphs to those that satisfy the good condition. In other words, the equivalence on flow graphs induced by the denotation of interfaces with respect to a good condition is also a congruence on $\bullet$.

Lemma 4.18. *For all good conditions* good *and interfaces $I = I_1 \oplus I_2$, if $H_1 \in [\![I_1]\!]_{\mathrm{good}}$ and $H_2 \in [\![I_2]\!]_{\mathrm{good}}$ then $H_1 \bullet H_2$ is defined and in $[\![I]\!]_{\mathrm{good}}$.*

When reasoning about programs, we will sometimes need to modify a flow graph region in a way that will increase the set of inflows. For instance, if we were to add a node n to a cyclic list, then the equivalence class of inflows will now also contain inflows that have n as a source. As the flow map is defined from all source nodes, this modification will also change the flow map. However, as long as the flow map from the existing sources is preserved, then the modified flow graph should still be able to compose with any context. Formally, we say an interface (In, a, f) is *contextually extended* by (In', a', f'), written $(In, a, f) \precsim (In', a', f')$, if and only if $In \subseteq In'$ and

$$\forall n \in \mathsf{dom}_1(f).\ \exists in \in In.\ in(n) \neq 0 \Rightarrow f(n, \cdot) = f'(n, \cdot).$$

where $f(n, \cdot)$ is the function that maps a node n' to $f(n, n')$. The following theorem states that contextual extension preserves composability and is itself preserved under interface composition.

Theorem 4.19 (Replacement). *If $I = I_1 \oplus I_2$, and $I_1 \precsim I_1'$, then there exists $I' = I_1' \oplus I_2$ such that $I \precsim I'$.*

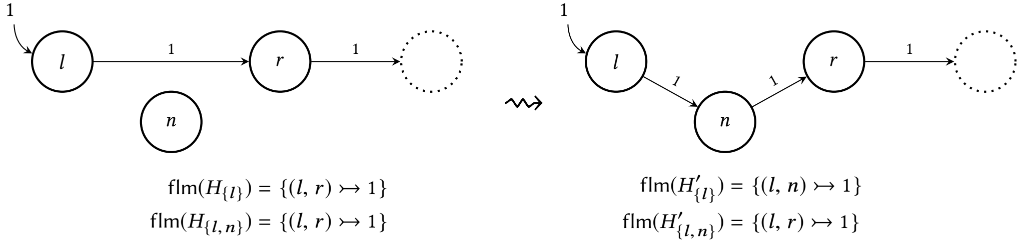

Fig. 3. Inserting a new node n into a list H between existing nodes l and r obtaining a new list H'. Edges are labeled with edge labels for path counting, and the nodes are labeled with their names. The flow maps of certain subgraphs before and after the modification are shown below.

5 A CONCURRENT SEPARATION LOGIC WITH FLOW INTERFACES

We next show how to extend a concurrent separation logic with flow interfaces. We use RGSep for this purpose as it suffices to prove the correctness of interesting programs, yet is relatively simple compared to other CSL flavors. However, many concurrent separation logics, including RGSep, are parametric in the separation algebra over which the semantics of programs is defined. So the technical development in this section can be easily transferred to these other logics.

RGSep is parametric in the program states (states must form a separation algebra), the language of the assertions (the original paper used the standard variant of separation logic), and the basic commands of the programming language (their semantics need to satisfy a *locality* property to obtain the frame rule of separation logic). For the most part, we piggyback on the development in [Vafeiadis 2008]. We denote the resulting logic by RGSep[FI]. Once we have shown that RGSep[FI] is a valid instantiation of RGSep, the soundness of all RGSep proof rules immediately carries over to RGSep[FI]. The only extra work that we will need to do before we can apply the logic is to derive a few generic lemmas for proving entailments between assertions that involve flow interfaces.

5.1 Reasoning about the Actual Heap

We first discuss the insert procedure on a singly-linked list as an example to motivate the design of our logic. The insert procedure first traverses the list to find two adjacent nodes l and r, with some appropriate properties, between which to insert the given node n. It then sets n's `next` field to point to r, and finally swings l's `next` pointer to n. A pictorial representation of the crucial step in this procedure is shown in the top portion of Fig. 3.

Using flow graphs directly as our program state, as presented thus far, poses three key challenges. Firstly, a key assumption SL makes on the programming language is that commands are *local* [Reynolds 2002]: informally, that if a command runs successfully on a state σ_1 and we run it on a composed state $\sigma_1 \cdot \sigma_2$ then it leaves σ_2 unchanged. However, even a basic operation on a flow graph, for instance changing the edge (l, r) to an edge (l, n), can potentially change the flows of all downstream nodes and is hence not local. We have seen that if we can find a region around the modification where the flow map is preserved, then the change will not affect any nodes outside the region. But which region do we choose as the footprint of a given command? Secondly, the programs that we wish to verify may temporarily violate the good condition during an update or a maintenance operation. For example, at the state on the left of Fig. 3 the new node n has no paths from the root, which violates the good condition describing a list. Thus we also need a way to describe these intermediate states. Finally and most immediately, the programs that we wish to verify operate on a program state that are more akin to standard heap representations than to flow graphs. We thus need a way to abstract the heap to a flow graph.

Our solution to all three problems is based on using the product of heaps and flow graphs as program states. The heap component represents the concrete state, and the standard commands such as heap allocation or mutation affect only the heap. Thus, these standard commands inherit locality from the standard semantics. The graph component is a ghost state and serves as an abstraction of the heap. Since standard commands only modify the heap, this abstraction may not be up-to-date with the heap. We add a new ghost command to *sync* a region of the graph with the current state of the heap, i.e. change the graph component to one that abstracts the heap component (using some abstraction as described below). We say the resulting region of the state is *in sync*. This **sync** command takes as an argument the new interface of the heap region to sync, and is only successful if the new interface contextually extends the old interface of that region. For instance, in Fig. 3, we cannot sync the region $\{l\}$ as l's outgoing edges have changed, we would have to sync the region $\{l, n\}$ (note that n is not a source node in this subgraph). Recall that Theorem 4.19 guarantees that this new region will still compose with the old context, making **sync** a local command and solving our first problem.

The fact that the graph portion of the state is not up-to-date with the heap also gives us a way to solve the problem of temporary violations of the good condition. When executing a series of commands, we delay syncing the graph portion of the state until the point when the good condition has been re-established. Thus, the graph component can be thought of as an abstraction of the heap at the last time it was in a good state. In the concurrent setting, one may wonder if we might expose such *dirty* regions (where the graph is not in sync with the heap) to other threads; however, if a thread is breaking some data structure invariants in a region, then it must be locked or marked off from interference from other threads in some manner. We add a *dirty predicate* $[\phi]_I$ to our logic to describe a region where the heap component satisfies ϕ but the flow graph component is potentially out of sync and satisfies the interface I. So to verify the list insert procedure, we would use a dirty predicate to describe l and n until we have modified both edges and are ready to use **sync**.

Finally, to solve the problem of relating the concrete heap to the abstract graph, we make use of the good condition. We define good by means of an SL predicate γ that not only describes the good condition on the flow in the graph component, but also describes the relation between a good node in the graph and its representation on the heap. For instance, here is the γ that we use to describe a singly-linked list:

$$\gamma(n, in, _, f) := \exists n'.\ n \mapsto n' \wedge in(n) = 1 \wedge f = \mathsf{ITE}(n' = null, \epsilon, \{(n, n') \rightarrowtail 1\}).$$

Note that this is essentially $\text{good}_{\mathsf{ls}}$ from §4.2 with the additional constraint that the edge label in the graph depends on n', the next pointer of n on the heap. We will show how to define a good graph predicate $\mathrm{Gr}(I)$ using such a γ to ensure that the heap and the graph are in sync.

5.2 Program States

We model the heap h as usual in standard separation logic. That is, the heap is a partial map from addresses Addr to values Val. Values include addresses, as well as the flow and node label domains and some auxiliary values for representing ghost state related to flow interfaces:

$$\mathsf{Val} \supseteq \mathsf{Addr} \cup D \cup A \cup (\mathsf{Addr} \rightharpoonup D) \cup \mathsf{IN} \cup (\mathsf{Addr} \times \mathsf{Addr} \rightharpoonup D) \cup \mathsf{FI}$$

A state augments the heap with an auxiliary ghost state in the form of a flow graph H that maintains an abstract view of the heap. Each graph node $n \in \mathsf{dom}(H)$ is an abstraction of potentially multiple heap nodes, one of which is chosen as the representative n. That is, we require $\mathsf{dom}(H) \subseteq \mathsf{dom}(h)$. Also, we have a node map r that labels each heap node with the graph node that abstracts it. This map is partial, i.e., we allow heap nodes that are not described by the flow graph.

$$
\begin{array}{l}
T ::= x \mid (x_1, x_2) \mid T_d \mid T_a \mid T_{in} \mid T_{In} \mid T_f \mid T_I \quad T_a ::= a \mid A \mid T_a \sqcup T_a \quad v, x, d, in, In, f, a, I \in \mathsf{Var} \\
T_d ::= d \mid D \mid T_{in}(x) \mid T_f(x_1, x_2) \mid T_d + T_d \mid T_d \cdot T_d \qquad T_{in} ::= in \mid \epsilon \mid \{x \rightarrowtail T_d, \dots\} \mid T_{in} + T_{in} \mid \mathbf{0} \\
T_{In} ::= In \mid \{T_{in}\} \quad T_f ::= f \mid \epsilon \mid \{(x_1, x_2) \rightarrowtail T_d, \dots\} \mid T_f + T_f \mid \mathbf{0} \quad T_I ::= I \mid (T_{In}, T_a, T_f) \mid T_I \oplus T_I \\
P ::= T = T \mid T_d \sqsubseteq T_d \mid T_a \sqsubseteq T_a \mid x \in T_I \mid (x_1, x_2) \in T_f \mid T_{in} \in T_{In} \mid T_I \precsim T_I \\
\phi ::= P \mid emp \mid true \mid x \mapsto T \mid \mathsf{Gr}(I) \mid [\phi]_I \mid \phi * \phi \mid \phi \mathrel{-\!\!*} \phi \mid \phi \land \phi \mid \exists v.\, \phi \mid \neg \phi
\end{array}
$$

Fig. 4. Syntax of RGSep[FI] terms and assertions.

Definition 5.1. A *state* $\sigma = (h, H, r) \in \mathsf{State}$ consists of a heap $h \colon \mathsf{Addr} \rightharpoonup \mathsf{Val}$, a flow interface graph $H \in \mathsf{FG}$ such that $\mathsf{dom}(H) \subseteq \mathsf{dom}(h)$, and a node map $r \colon \mathsf{dom}(h) \rightharpoonup \mathsf{dom}(H)$ such that $r|_{\mathsf{dom}(r) \cap \mathsf{dom}(H)} = (\lambda x. x)$.

The composition operator on states is a partial function $\sigma_1 \cdot \sigma_2$, defined as

$$
(h_1, H_1, r_1) \cdot (h_2, H_2, r_2) \coloneqq \begin{cases} (h_1 \cdot h_2, H_1 \bullet H_2, r_1 \uplus r_2) & \text{if } h_1 \cdot h_2 \text{ and } H_1 \bullet H_2 \text{ defined} \\ \text{undefined} & \text{otherwise.} \end{cases}
$$

Here, $h_1 \cdot h_2$ denotes the standard disjoint union of heaps. The empty state σ_e is defined as $\sigma_e \coloneqq (h_e, H_{\mathrm{e}}, r_e)$ where h_e and r_e are the empty heap and empty node map, respectively. The fact that states form a separation algebra easily follows from Theorem 4.14.

Theorem 5.2. *The structure* $(\mathsf{State}, \cdot, \sigma_e)$ *is a separation algebra.*

For simplicity we do not use the variable as resource model [Bornat et al. 2006] for the treatment of stack variables. Instead, we follow [Vafeiadis 2008] and assume that all stack variables are local to a thread, and that all global variables are read-only. Stack variables are allocated on the local heap and global variables on the shared heap. We assume that each program variable x is located at a fixed address &x. In all assertions occurring in program specifications throughout the rest of the paper we have an implicit &x $\mapsto x$ for every program variable x in scope.

5.3 Assertions

The syntax of assertions is summarized in Fig. 4. We assume an infinite set of (implicitly sorted) logical variables Var. The meta variable v stands for a variable of an arbitrary sort and the meta variable x for a variable of a sort denoting addresses. The sorts of other variables are as expected. Assertions are built from sorted terms – one sort per semantic domain of interest. The terms T_d denote flow domain values, T_a node label values, etc. For inflow terms T_{in}, the constant ϵ denotes the empty inflow in_{e}, $T_{in} + T_{in}$ denotes the lifting of + to partial functions[2], and the constant $\mathbf{0}$ denotes the total inflow that maps all addresses to 0. Terms for flow maps are constructed similarly.

The atomic pure predicates P include equalities, domain membership tests for interfaces, flow maps, and inflow equivalence classes, as well as contextual extension of flow interfaces. The actual separation logic assertions ϕ include the standard SL assertions as well as the extended assertions $\mathsf{Gr}(I)$ and $[\phi]_I$ related to flow interfaces.

The good graph predicate $\mathsf{Gr}(I)$ describes a heap region that is abstracted by a good flow graph satisfying the flow interface I. The semantics of the logic is parametric in what constitutes a good graph. For simplicity, we consider only one type of good graph predicate, i.e., all instances of good graph predicates imply the same data structure invariant.

[2] $(f + g)(x) \coloneqq \mathsf{ITE}(x \in \mathsf{dom}(f), \mathsf{ITE}(x \in \mathsf{dom}(g), f(x) + g(x), f(x)), \mathsf{ITE}(x \in \mathsf{dom}(g), g(x), \text{undefined}))$

$$
\begin{array}{rlcl}
(h, H, r), i & \models T \in T_{I,f} & \Leftrightarrow & [\![T]\!]_i \in \mathsf{dom}([\![T_{I,f}]\!]_i) \\
(h, H, r), i & \models T_{in} \in T_{In} & \Leftrightarrow & [\![T_{in}]\!]_i \in [\![T_{In}]\!]_i \\
(h, H, r), i & \models T_I \precsim T'_I & \Leftrightarrow & [\![T_I]\!]_i \precsim [\![T'_I]\!]_i \\
(h, H, r), i & \models emp & \Leftrightarrow & h = h_e \land H = H_{\mathrm{e}} \\
(h, H, r), i & \models x \mapsto T & \Leftrightarrow & \mathsf{dom}(h) = \{[\![x]\!]_i\} \land h([\![x]\!]_i) = [\![T]\!]_i \land H = H_{\mathrm{e}} \\
(h, H, r), i & \models \mathsf{Gr}(I) & \Leftrightarrow & \mathsf{dom}(h) = \mathsf{dom}(r) \land H \in [\![[\![I]\!]_i]\!]_{\mathsf{good}_{h,r}} \\
(h, H, r), i & \models [\phi]_I & \Leftrightarrow & \mathsf{dom}(h) = \mathsf{dom}(r) \land H \in [\![[\![I]\!]_i]\!] \land \exists H', r'.\ (h, H', r'), i \models \phi \\
\sigma, i & \models \phi_1 * \phi_2 & \Leftrightarrow & \exists \sigma_1, \sigma_2.\ (\sigma = \sigma_1 \cdot \sigma_2) \land (\sigma_1, i \models \phi_1) \land (\sigma_2, i \models \phi_2) \\
 & & \dots &
\end{array}
$$

Fig. 5. Semantics of assertions.

We also need to permit certain regions of the heap that are under modification to be not good. These regions are described using the dirty predicate $[\phi]_I$. This describes a state where the flow graph satisfies I but the heap satisfies ϕ.

We use some syntactic shorthands for RGSep[FI] assertions. For instance, $\mathsf{N}(x, I)$ denotes a singleton good graph $\mathsf{Gr}(I)$ containing x, and the separating inclusion operator $\phi_1 \twoheadrightarrow\!\!* \ \phi_2$ defined as $(\phi_1 * true) \land \phi_2$. For an interface $I = (In, a, f)$ we write I^{In}, a^a and I^f for In, a and f. The rest will be defined as and when they are used.

Terms are interpreted with respect to an interpretation $i\colon \mathsf{Var} \to \mathsf{Val}$ that maps the logical variables to values. For a variable $v \in \mathsf{Var}$, we write $[\![v]\!]_i$ to mean $i(v)$ and also extend this function to terms T. We omit the definition of this partial function as it is straightforward. Note that some terms, for instance interface composition $T_I \oplus T_I$, may be undefined; we define the semantics of any atom containing an undefined term to be false. The semantics of assertions ϕ is given by the satisfaction relation $\sigma, i \models \phi$ and we denote the induced entailment relation by $\phi_1 \models \phi_2$. The satisfaction relation is defined in Fig. 5. Again, we omit some of the obvious cases. Most of the cases are straightforward. In particular, the standard SL assertions have their standard semantics. We only discuss the cases for $\mathsf{Gr}(I)$ and $[\phi]_I$.

The predicate $\mathsf{Gr}(I)$ describes a region whose heap is abstracted by a good flow graph. We tie the concrete representation in the heap h to the flow graph H of the state using the good node condition, which we assume is defined by an SL predicate $\gamma(x, in, a, f)$. This predicate specifies the heap representation of a node x in the graph as well as any invariants that x must satisfy on $\mathsf{flow}(H)(x)$. We restrict γ to standard SL assertions, i.e., γ is not allowed to include occurrences of the predicates $\mathsf{Gr}(I)$ and $[\phi]_I$.

The predicate γ implicitly defines a good node condition $\mathsf{good}_{h,r}$ on flow interfaces:

$$\mathsf{good}_{h,r}(n, in, a, f) :\Leftrightarrow (h|_{r^{-1}(n)}, H_{\mathrm{e}}, r_e), \epsilon \models \gamma(n, in, a, f)$$

The semantics of $\mathsf{Gr}(I)$ then uses this condition to tie the flow graph H to the heap h and ensures that each node in the graph satisfies its local invariant on the flow. The semantics of $[\phi]_I$ is slightly more complicated. It states that the current flow graph H satisfies I and there is some flow interface graph H' that satisfies ϕ in the current heap.

A precise assertion is one that if it holds for any substate, then it holds for a unique substate. To use our new predicates in actions to describe interference, we require them to be precise.

Lemma 5.3. *The graph predicates* $\mathsf{Gr}(I)$ *and* $[\phi]_I$ *are precise.*

$$\{\&\mathtt{x} \mapsto _ * \exists y.\ \phi(y)\}\ \mathtt{x}\ :|\ \phi(\mathtt{x})\ \{\&\mathtt{x} \mapsto x * \phi(x)\} \qquad \{I \lesssim I' \land [\mathrm{Gr}(I')]_I\}\ \mathbf{sync}(\mathtt{I'})\ \{\mathrm{Gr}(I')\}$$

$$\{x \mapsto v * [\phi]_I \land y \in I\}\ \mathbf{mark}(\mathtt{x},\mathtt{y})\ \{[x \mapsto v * \phi]_I\} \qquad \{x = y \land x \mapsto v\}\ \mathbf{mark}(\mathtt{x},\mathtt{y})\ \{[x \mapsto v]_{(\{\{x \rightarrowtail 0\}\}, a_e, \epsilon)}\}$$

$$\left\{[x \mapsto v * \phi]_I \land x \notin I^{In}\right\}\ \mathbf{unmkark}(\mathtt{x})\ \{x \mapsto v * [\phi]_I\} \qquad \left\{[x \mapsto v]_I \land I^{In} = \{\{x \rightarrowtail 0\}\}\right\}\ \mathbf{unmkark}(\mathtt{x})\ \{x \mapsto v\}$$

Fig. 6. Specifications of ghost commands

5.4 Programming Language and Semantics

The programming language is very similar to the one used in RGSep and its semantics is also mostly identical to the one given in Section 3.2 of [Vafeiadis 2008]:

$$C \in \mathrm{Com} ::= \quad \mathbf{skip} \mid c \mid C_1; C_2 \mid C_1 + C_2 \mid C^* \mid \langle C \rangle \mid C \| C$$
$$c ::= \quad \mathtt{x}\ :|\ \phi(\mathtt{x}) \mid \mathbf{sync}(\mathtt{I}) \mid \mathbf{mark}(\mathtt{x}, \mathtt{y}) \mid \mathbf{unmkark}(\mathtt{x}) \mid \dots$$

The standard commands include the empty command (**skip**), basic commands, sequential composition, non-deterministic choice, looping, atomic commands, and parallel composition. Apart from the standard basic commands, we add several special ghost commands to the language, which we describe next. Their axiomatic specifications are given in Fig. 6.

The *wishful assignment* $\mathtt{x}\ :|\ \phi(\mathtt{x})$ is a ghost command that allows us to bring a witness to an existentially quantified assertion onto the stack. This helps us to get a handle on the updated flow interface of a modified region so that we can pass it to the **sync**. The special ghost command **sync** is used to bring a flow graph region back to sync with the heap. The **mark** command[3] is used to formally associate a (potentially new) heap node with the graph node that abstracts it. This is useful in cases where many heap nodes are abstracted by the same graph node. Similarly, **unmkark** removes the association between a heap and a graph node. Each command has two successful cases, one where the node being (un)marked is the representative of a graph node, and one where it is represented by another node.

The semantics of the new ghost commands are given in the companion technical report [Krishna et al. 2017]. The semantics of all other commands are inherited from [Vafeiadis 2008]. That is, these commands only affect the heap component of the state and leave the flow graph and node map components unchanged. If a standard command C modifies an unmarked heap location, then we can use the standard SL specification to reason about it. If it modifies a heap location that is marked, then we can lift C's SL specification to an RGSep[FI] specification using the following rule

$$\frac{\{\psi_1\}\ C\ \{\psi_2\}}{\{[\psi_1]_I\}\ C\ \{[\psi_2]_I\}}$$

where $\psi_$ denotes an assertion that does not contain the predicates $\mathrm{Gr}(I)$ and $[\phi]_I$. We shall see generic lemmas to convert a good graph node into a dirty region in §5.5.

We show in [Krishna et al. 2017] that the new ghost commands are local. Our logic RGSep[FI] thus satisfies all the requirements for a correct instantiation of RGSep.

5.5 Proving Entailments

Finally, we show several generic lemmas for proving entailments about SL assertions with flow interfaces. We only focus on lemmas that involve flow interfaces and omit lemmas about simple pure assertions involving inflows and flow maps. We also omit lemmas about the algebraic properties of flow interface composition, such as associativity. The lemmas are shown in Fig.7.

[3] not to be confused with the notion of marking logically deleted nodes in Harris' list

$$\begin{array}{rcll}
\mathsf{Gr}(I) \land x \in I & \models & \exists I_1, I_2.\ \mathsf{N}(x, I_1) * \mathsf{Gr}(I_2) \land I = I_1 \oplus I_2 & (\textsc{Decomp}) \\
(\mathsf{Gr}(I_1) * \mathit{true}) \land \mathsf{Gr}(I) & \models & \exists I_2.\ \mathsf{Gr}(I_1) * \mathsf{Gr}(I_2) & (\textsc{GrDecomp}) \\
\left[Q_1(I_1')\right]_{I_1} * \left[Q_2(I_2')\right]_{I_2} \land I' = I_1' \oplus I_2' & \models & \left[Q_1(I_1') * Q_2(I_2')\right]_{I_1 \oplus I_2} & (\textsc{Comp}) \\
\mathsf{Gr}(I_1) * \mathsf{Gr}(I_2) & \models & \mathsf{Gr}(I_1 \oplus I_2) & (\textsc{GrComp}) \\
(\mathsf{Gr}(I_1) * \mathsf{Gr}(I_2)) \land (\mathsf{N}(x, I_x) * \mathsf{Gr}(I_3)) & \models & \exists I_4.\ \mathsf{Gr}(I_1) * \mathsf{N}(x, I_x) * \mathsf{Gr}(I_4) & (\textsc{Disj}) \\
\land\ {I_x}^a \not\subseteq {I_1}^a & & & \\
\mathsf{N}(x, I) & \equiv & \exists in, a, f.\ [\gamma(x, in, a, f)]_I \land in = \{x \rightarrowtail _\} & (\textsc{Conc}) \\
 & & \land\ I = (\{in\}, a, \mathsf{ITE}(in(x) = 0, \epsilon, f)) & \\
[\mathsf{N}(x, I')]_I & \equiv & \exists in, a, f.\ [\gamma(x, in, a, f)]_I \land in = \{x \rightarrowtail _\} & (\textsc{Abs}) \\
 & & \land\ I' = (\{in\}, a, \mathsf{ITE}(in(x) = 0, \epsilon, f)) & \\
[\mathit{true}]_I \land [\mathit{true}]_{I'} & \models & I = I' & (\textsc{Uniq}) \\
I' = I \oplus (\{\{x \rightarrowtail 0\}\}, _, _) \land in \in I & \models & \exists in' \in {I'}^{In}.\ in + \mathbf{0} = in' + \mathbf{0} & (\textsc{AddIn}) \\
I' = I \oplus (_, _, \epsilon) \land I^f = \epsilon & \models & {I'}^f = \epsilon & (\textsc{AddF}) \\
I_1 \precsim I_1' & \models & (I_1 \oplus I_2) \precsim (I_1' \oplus I_2) & (\textsc{Repl}) \\
I \precsim I' \land in \in I & \models & in \in {I'}^{In} & (\textsc{ReplIn}) \\
I \precsim I' \land I^f = \epsilon & \models & {I'}^f = \epsilon & (\textsc{ReplF}) \\
I = I_1 \oplus I_2 \land (x, y) \in {I_1}^f \land I^f = \epsilon & \models & y \in I_2 & (\textsc{Step})
\end{array}$$

Fig. 7. Generic lemmas for proving entailments

For example, the lemma (Decomp) implies that we can pull an arbitrary node $\mathsf{N}(x, I_1)$ from a good graph $\mathsf{Gr}(I)$ containing x, obtaining a flow interface constraint $I = I_1 \oplus I_2$ that remembers that the two parts composed to I. Lemma (Comp) uses $Q(I)$ to denote either $\mathsf{Gr}(I)$ or $[\phi]_I$ for some ϕ and tells us that we can combine two dirty regions if the interfaces they are expecting (I_1' and I_2') are compatible. The lemma (GrComp) states that we can always abstract two good graphs $\mathsf{Gr}(I_1) * \mathsf{Gr}(I_2)$ to a larger good graph $\mathsf{Gr}(I)$ that satisfies the composite interface $I = I_1 \oplus I_2$. The validity of this entailment follows from Lemma 4.18. (Conc) and (Abs) allow us to shift our reasoning from graph nodes to their heap representation and back. (AddIn) and (AddF) can be used to reason about adding new nodes to the state.

One advantage of our logic is that a lemma like (GrComp) is very general and can be used to compose two lists, two sorted lists, or two min-heaps. The good condition will ensure, for instance, that the composition of sorted lists is itself sorted.

6 APPLICATION 1: THE HARRIS LIST

We now demonstrate our flow framework by describing and verifying the Harris list. For simplicity of presentation, we consider a data structure without keys and abstract the algorithm to one that non-deterministically chooses where to insert a node or which node to delete. This is without loss of generality, as we are only proving memory safety and absence of memory leaks. Our proof can be extended to prove these properties on the full Harris list, and our framework can also prove the key invariants needed to prove linearizability.

```
1  procedure insert() { {⌈Φ⌉}
2    // Where: Φ := ∃I. Gr(I) ∧ φ(I),   φ(I) := ∃in ∈ I^in. in + 0 = {mh ↣ (1, 0)} + {fh ↣ (0, 1)} + 0 ∧ I^f = ϵ
3    var l := mh; {⌈N(l, I_l) −⊛ Φ⌉ ∧ l = mh}
4    var r := getUnmarked(l.next);
5    while (r != null && nondet()) { {⌈(N(l, I_l) ∗ N(r, I_r)) −⊛ Φ⌉}
6      l := r; r := getUnmarked(l.next); {⌈(N(l, I_l)) −⊛ Φ⌉ ∧ (r ≠ null ⇒ ⌈(N(l, I_l) ∗ N(r, I_r)) −⊛ Φ⌉)}
7    }
8    if (!isMarked(r)) { {⌈N(l, I_l) −⊛ Φ⌉ ∧ ¬M(r)}
9      var n := new Node(r, null); {⌈N(l, I_l) −⊛ Φ⌉ ∗ n ↦ r, null ∧ ¬M(r)}
10     mark(n, n); {⌈N(l, I_l) −⊛ Φ⌉ ∗ [n ↦ r, null]_{I_n} ∧ ¬M(r)} // Where: I_n := ({n ↣ (0, 0)}, ⊥, ϵ)
11     atomic {  // CAS(l.next, r, n)
12       if (l.next == r) {
13         {([l ↦ r, _]_{I_l} ∧ N(l, I_l)) ∗ [n ↦ r, null]_{I_n} ∗ Gr(I_2) ∧ I_l^a = ◇ ∧ I′ = I_l ⊕ I_n ⊕ I_2 ∧ φ(I′)}
14         l.next := n;
15         {[N(l, I′_l) ∗ N(n, I′_n)]_{I_1} ∗ Gr(I_2) ∧ I′_l^a = ◇ ∧ I′_n^a = ◇ ∧ I′ = I_l ⊕ I_n ⊕ I_2 ∧ φ(I′) ∧ I_1 = I_l ⊕ I_n ≈ I′_l ⊕ I′_n}
16         var I′_1 :| [Gr(I′_1)]_{I_1} ∧ {l ↣ _, n ↣ _} ∈ I_1^In;
17         sync(I′_1); {N(l, I′_l) ∗ N(n, I′_n) ∗ Gr(I_2) ∧ I′_l^a = ◇ ∧ I′_n^a = ◇ ∧ I″ = I′_l ⊕ I′_n ⊕ I_2 ∧ φ(I″)}
18         var b := true; {(N(l, I′_l) ∗ N(l, I′_l)) −⊛ Φ ∧ I′_l^a = ◇ ∧ I′_n^a = ◇ ∧ b}
19       } else var b := false; {⌈N(l, I_l) −⊛ Φ⌉ ∗ [n ↦ r, null]_{I_n} ∧ ¬b}
20     } ... // If CAS failed, unmark and free n, and call insert() again
21   }
22 }
```

Fig. 8. A fragment from the insert procedure on the Harris list.

We now describe the flows and good conditions we use for this proof. To describe the structural properties, we use the product of two path-counting flows, one counting paths from the head of the main list mh and one from the head of the free list fh. To reason about marking, we use the node domain $\mathbb{N} \cup \{\diamond, \top\}$ under the ordering where $\diamond$ is the smallest element, $\top$ is the largest element, and all other elements are unordered. We label unmarked nodes with $\diamond$, and marked nodes with the thread ID $t \in \mathbb{N}$ of the thread that marked the node. This is in order to enforce that only the thread that marks a node may link it to the free list, a property needed to prove that the free list is acyclic.

Our good condition is specified by the following predicate:

$$\begin{aligned}\gamma(n, in, a, f) &:= \exists n', n''.\ n \mapsto n', n'' \wedge a \neq \top \wedge (M(n') \Leftrightarrow a \neq \diamond) \wedge (0,0) < in(n) \leq (1,1) \\ &\wedge (in(n) \geq (0,1) \Rightarrow a \neq \diamond) \wedge (n = ft \Rightarrow in(n) \geq (0,1)) \wedge (in(n) \leq (1,0) \Rightarrow n'' = null) \\ &\wedge f = \mathsf{ITE}(u(n') = null, \epsilon, \{(n, u(n')) \rightarrowtail (1,0)\}) + \mathsf{ITE}(n'' = null, \epsilon, \{(n, n'') \rightarrowtail (0,1)\}).\end{aligned}$$

This condition expresses that every node n is a heap cell containing two pointers n' (for the `next` field) and n'' (for the `fnext` field). n is either unmarked or marked with a thread ID ($a \neq \top$), but only if the value n' of field `next` has its mark bit set (encoded using the predicate M). We next use the path-counting flows to say that n has exactly 1 path on `next` edges from mh, on `fnext` edges from fh, or both. This enforces that all nodes are in at least one of the two lists, and this is how we establish absence of memory leaks. The next few conjuncts say that all nodes in the free list are marked ($in \geq (0,1) \Rightarrow a \neq \diamond$), that ft is a node in the free list, and that main list nodes have no

`fnext` edges. The final line describes the edges: n has a `next` edge (encoded with the edge label $(1, 0)$) to $u(n')$ (the unmarked version of n', i.e. the actual pointer obtained from n' by masking the mark bit), and an `fnext` edge (label $(0, 1)$) to n'', but only if they are not null.

We use the global data structure invariant Φ shown in line 2 of Fig. 8, which gives the appropriate non-zero inflow to fh and mh. It is easy to see that with good condition γ, Φ describes a structure satisfying properties (a) to (d) of the Harris list from §2.

We now describe the RGSep[FI] proof for the `insert` procedure, shown in Fig. 8. Variables that are not program variables in the annotations are implicitly existentially quantified. All the entailments in this proof sketch can be proved with the help of our library of lemmas from §5.5.

The procedure starts in a state satisfying Φ and sets a variable `l` to equal the head of the main list. Since `l` is in the domain of the inflow I^{in} we use (Decomp) to decompose Φ and get a single node $(\mathsf{N}(l, I_l))$ included in the larger graph satisfying Φ (line 3). We then read `l.next`, store the unmarked version in `r`, and enter a loop. At the beginning of the loop (line 5), since $r \neq null$, we know by γ that `l` has a `next` edge to r. But since Φ implies $I^f = \epsilon$, we can use (GrDecomp), (Step), and (Decomp) to extract the node corresponding to r and obtain the annotation on line 5.

Inside the loop, we move `l` to `r` and then again read `l.next`, unmark it, and set it to `r` (line 6). This dereference is memory safe because we know that we have access permission to `l` (by $\mathsf{N}(r, I_r)$ on line 5). Finally, to establish the annotation on line 6, we "drop" the node predicate corresponding to the old value of `l` and absorb it into Φ. The second conjunct is derived similar to the annotation on line 5. Note that the annotation at the end of the loop implies the annotation at the beginning of the loop if the loop check succeeds.

The loop terminates when the non-deterministic loop condition fails – presumably at the correct position to insert the new node. Since we only want to insert new nodes into the main list, we ensure `l` is unmarked (line 8). We then create the new node with `next` field r (which may equal null) in line 9. As this allocation only modifies the heap, the resulting node is described using a standard SL points-to predicate in the local state of the thread. To create a corresponding node in the graph we use the **`mark`** ghost command, which creates a new graph node with a zero inflow, the bottom node label, and no outgoing edges. The resulting state is described with a dirty predicate (line 10) as it is neither in sync nor a good node.

The next step is to swing l's `next` pointer from r to n using a CAS operation. The CAS is expanded into an atomic block (lines 11 to 20) in order to show the intermediate proof steps. Note that inside the atomic block there is no need for separation of shared and local state, as in RGSep one can reason sequentially about atomic executions. If the compare portion of the CAS succeeds, then we know that l is unmarked (${I_l}^a = \diamond$) and its next field equals r (we use a dirty predicate around l to express this). We use (GrDecomp), (Comp), (AddIn), and (AddF) to decompose Φ at this point and infer that I', the global interface extended with n, also satisfies the global conditions $\varphi(I')$ (line 13). The modification to `l.next` in the next line is memory safe since we have access permission to l (by $l \mapsto r, _$).

Before we can bring the graph abstraction back to sync, we must establish that the change to the interface of some region is a contextual extension. As we saw with the example of inserting in a singly-linked list in Fig. 3, we must consider the region containing $\{l, n\}$. Some pure reasoning about path-counts in this region can be used to infer that $I_l \oplus I_n \approx I'_l \oplus I'_n$ (line 15)[4]. We then use a wishful assignment (line 16) to take a snapshot of the new interface $\mathtt{I}'_1$ of the region we wish to sync.

The other fact to prove before syncing the graph is to show that the region under modification is itself a good state. This is indicated in line 15 by using good node predicates inside the dirty

[4] $I \approx I'$ is shorthand for $I \precsim I' \land I' \precsim I$.

region. To establish that n is a good node, for instance, we use a new interface I'_n which gives it an inflow of $(1, 0)$, a node abstraction of $\diamond$, and a flow map of one edge to r with label $(1, 0)$. We then check that the heap representation of n along with this interface satisfies γ. Similarly, we check that l, with its new edge to n is still a good node. As the entire dirty region is a good state with a contextually extended interface, we can now use **sync** (line 17) to update the graph. We can then use the (Repl), (ReplIn), (ReplF), and (Comp) rules to establish that the new global interface satisfies the invariant $\varphi(I'')$.

The final state (line 18) is once again a good state, which means that we have shown both memory safety and absence of memory leaks of this procedure. We have omitted the rest of the insert procedure due to space constraints, which frees the node n and restarts if the CAS failed, but note that this can be proved in a similar fashion. We have also omitted the reasoning about interference by other threads, which is done in RGSep by checking that each intermediate assertion is stable under the action of other threads. This is a syntactic check that can be done with the help of the entailment lemmas presented in Fig. 7. The full proof can be found in the technical report [Krishna et al. 2017, §A]. One can also prove linearizability – for the Harris list, this requires a technique such as history variables since linearization points are dynamic – but the invariants of the data structure needed to show linearizability are expressible using flows.

This example shows that reasoning about programs using flow graphs and interfaces is as natural as with inductive predicates, and we can use similar unrolling and abstraction lemmas to reason about traversals of the data structure. Note that the intermediate assertions would have looked almost identical if we did not have a free list – the only place where we reason about the free list is in the global interface on line 2 and when we prove the local property $I_l \oplus I_n \approx I'_l \oplus I'_n$ on line 15.

7 APPLICATION 2: DICTIONARIES

In this section, we use the flow framework to verify a large class of concurrent dictionary implementations. We base our approach on the *edgeset framework* of Shasha and Goodman [1988] that provides invariants, in terms of reachability properties of sets of keys, for proving linearizability of dictionary operations (search, insert, and delete). Linearizability means, informally, that each operation appears to happen atomically, i.e. at a single instant in time, and that if operation o_1 finishes before o_2 begins, then o_1 will appear to happen before o_2.

By encoding the edgeset framework using flows, we obtain a method of proving linearizability as well as memory safety. More importantly, we can use the power of flows to encode the data constraints independently of the shape. Thus our encoding will be data-structure-agnostic, meaning that we can verify *any* dictionary implementation that falls within the edgeset framework.

We first briefly describe the original edgeset framework, then show how to encode it using flows. We then give an abstract algorithm template and specifications for the dictionary operations that can be instantiated to concrete implementations. In the technical report [Krishna et al. 2017, §B] we describe such an instantiation to a nontrivial implementation based on B+ trees, and show how we can verify it using this framework.

7.1 The Edgeset Framework

A dictionary is a key-value store that implements three basic operations: search, insert, and delete. For simplicity of exposition, we ignore the data values and treat the dictionary as containing only keys. We refer to a thread seeking to search for, insert, or delete a key k as an operation on k, and to k as the operation's query key. Let KS be the set of possible keys, e.g., all integers.

The edgeset framework describes certain invariants on the data layout in dictionary implementations and an abstract algorithm that will be correct if the invariants are maintained. These invariants do not specify the shape of the data structure to be say a tree, list, or hash table, and

instead describe properties of an abstract graph representation of the data structure (rather like the flow graph). The nodes in the graph can represent, for instance, an actual heap node (in the case of a list), an array cell (in the case of a hash table), or even a collection of fields and arrays (in the case of a B-tree). Nodes are labeled with the set of keys stored at the node (henceforth, the contents of the node). Each edge is labeled by a set of keys that we call the *edgeset*, and is defined as follows.

When a dictionary operation arrives at a node n, the set of query keys for which the operation traverses an edge (n, n') is called the edgeset of (n, n'). For example, in a BST, an operation on k moves from a node to its left child n_l if k is less than the key contained at the node, hence the edgeset of (n, n_l) is $\{k \mid k < n.key\}$. Note that k can be in the edgeset of (n, n') even if n is not reachable from any root; the edgeset is the set of query keys for which an operation would traverse (n, n') assuming it somehow found itself at n.

The *pathset* of a path between nodes n_1 and n_2 is defined as the intersection of edgesets of every edge on the path, and is thus the set of keys for which operations starting at n_1 would arrive at n_2 assuming neither the path nor the edgesets along that path change. For example, in a sorted list, the pathset of a path from the head of a list to a node n is equal to the edgeset of the edge leading into n (i.e. the set $\{k \mid n'.key < k\}$ where n' is n's predecessor). With this, we define the *inset* of a node n as the union of the pathset of all paths from the root node to n. [5] If we take the inset of a node n, and remove all the keys in the union of edgesets of edges leaving n, we get the *keyset* of n.

There are three desirable conditions on graphs representing dictionary data structures:

(GS1) The keysets of two distinct nodes are disjoint[6].
(GS2) The contents of every node are a subset of the keyset of that node.
(GS3) The edgesets of two distinct edges leaving a node are disjoint.

Intuitively, (GS1) and (GS2) tell us we can treat the keyset of n as the set of keys that n can potentially contain. In this case, k is in the inset of n if and only if operations on k pass through n, and k is in the keyset of n if and only if operations on k end up at n. (GS3) requires that there is a deterministic path that operations follow, which is a desirable property that is true of all data structures in common use. A dictionary state that satisfies these conditions is called a *good state*.

The Keyset Theorem of Shasha and Goodman [1988] states, informally, that if every atomic operation preserves the good state property and k is in the keyset of a node n at the point when the operation looks for, inserts, or deletes k at n, then the algorithm is linearizable. Intuitively, if k is in the keyset of n, then since the keysets are disjoint we know that no other thread is performing an operation on k at any other node. And once this operation acquires a lock on n (or establishes exclusive access in another way, e.g. through a compare and swap), we know that operations on n will be atomic.

The challenge with using this framework for a formal proof in an SL-based program logic is that the invariants depend on quantities, like the inset, that are not local. The inset of a node n depends both on the global root as well as all paths in the data structure from the root to n. We show next how to convert the good state conditions into local properties of nodes using flows.

7.2 Encoding the Edgeset Framework using Flows

Given a potentially-infinite set of keys, KS, the set of subsets of KS forms a flow domain: $(2^{\mathsf{KS}}, \subseteq, \cup, \cup, \cap, \emptyset, \mathsf{KS})$. The node domain must contain a set of keys to keep track of the contents of each node, but we also wish to reason about locking. The challenge here is that threads may modify a locked node using a series of atomic operations such that the node does not satisfy the good state

[5]If there are multiple roots, then each edge has a different edgeset depending on the root, and hence the definitions of pathset and inset also depend on the particular root. The formalism can be extended to handle multiple roots in this manner.
[6]The original paper required the keysets to partition KS, but we note that this weaker condition is sufficient for linearizability.

```
1  procedure dictionaryOp(Key k) {  {[Φ]}
2    var c := r;  {[N(c, I_c) -** Φ]}
3    while (true) {  {[N(c, I_c) -** Φ]}
4      lock(c);  {[N(c, I_c) -** Φ] ∧ I_c^a = (_, {t})}
5      var n;
6      if (inRange(c, k)) {  {[N(c, I_c) -** Φ] ∧ I_c^a = (_, {t}) ∧ k ∈ I_c^In(c)}
7        n := findNext(c, k);  {[N(c, I_c) -** Φ] ∧ I_c^a = (_, {t}) ∧ k ∈ I_c^In(c)
                                ∧ (n ≠ null ∧ k ∈ I_c^f(c, n) ∨ n = null ∧ ∀x ∈ I_c^f. k ∉ I_c^f(c, x))}
8        if (n == null) break;
9        {[(N(c, I_c) * N(n, I_n)) -** Φ] ∧ I_c^a = (_, {t})}
10     } else {
11       n := r;  {[N(c, I_c) -** Φ] ∧ I_c^a = (_, {t}) ∧ n = r}
12     }  {[(N(c, I_c) * N(n, I_n)) -** Φ] ∧ I_c^a = (_, {t}) ∨ [N(c, I_c) -** Φ] ∧ I_c^a = (_, {t}) ∧ c = n = r}
13     unlock(c);
14     c := n;  {[N(c, I_c) -** Φ]}
15   }  {[N(c, I_c) -** Φ] ∧ I_c^a = (_, {t}) ∧ k ∈ I_c^In(c) ∧ ∀x ∈ I_c^f. k ∉ I_c^f(c, x)}
16   var res := decisiveOp(c, k);  {[N(c, I_c) -** Φ] ∧ I_c^a = (_, {t})}
17   unlock(c);  {[N(c, I_c) -** Φ]}
18   return res;  {[Φ]}
19 }
```

Fig. 9. The give-up template, with proof annotations in our logic.

$\left\{\boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!* \Phi}\right\}$ `lock(c);` $\left\{\boxed{\mathsf{N}(c, I'_c) \twoheadrightarrow\!\!* \Phi} \land {I'_c}^a = (_, \{t\}) \land I_c \approx I'_c\right\}$

$\left\{\boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!* \Phi} \land {I_c}^a = (_, \{t\})\right\}$ `unlock(c);` $\left\{\boxed{\mathsf{N}(c, I'_c) \twoheadrightarrow\!\!* \Phi} \land I_c \approx I'_c\right\}$

$\left\{\boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!* \Phi} \land {I_c}^a = (_, \{t\})\right\}$ `res := inRange(c, k);` $\left\{\boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!* \Phi} \land (res \Rightarrow k \in {I_c}^{In}(c))\right\}$

$\left\{\boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!* \Phi} \land {I_c}^a = (_, \{t\})\right\}$ `n := findNext(c, k);` $\left\{\boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!* \Phi} \land (n \neq null \land k \in {I_c}^f(c, n) \lor n = null \land \forall x \in {I_c}^f.\ k \notin {I_c}^f(c, x))\right\}$

$\left\{\boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!* \Phi} \land {I_c}^a = (C, \{t\}) \land k \in {I_c}^{In}(c) \land \forall x \in {I_c}^f.\ k \notin {I_c}^f(c, x)\right\}$ `res := decisiveOp(c, k);` $\left\{\boxed{\mathsf{N}(c, I'_c) \twoheadrightarrow\!\!* \Phi} \land I_c \approx I'_c \land \Psi\right\}$

$$\text{where } \Psi := \begin{cases} {I'_c}^a = (C, \{t\}) \land res \Leftrightarrow k \in C & \text{for member} \\ {I'_c}^a = (C \cup \{k\}, \{t\}) \land res \Leftrightarrow k \notin C & \text{for insert} \\ {I'_c}^a = (C \setminus \{k\}, \{t\}) \land res \Leftrightarrow k \in C & \text{for delete} \end{cases}$$

Fig. 10. Specifications for helper functions.

conditions in between operations, thus making the Keyset Theorem inapplicable. We do not want to use the dirty predicate to reason about such nodes, as this will complicate the global shared state invariant. Instead, we label nodes with elements of $\mathbb{N} \uplus \overline{\mathbb{N}}$, where $\overline{\mathbb{N}} := \{\overline{x} \mid x \in \mathbb{N}\}$. Here, 0 denotes an unlocked node, t denotes a node locked by thread t, and $\overline{t}$ denotes a node locked by thread t whose heap representation is out of sync. Formally, we use a product of sets of keys and

sets of augmented thread IDs as the node domain: $(2^{\mathsf{KS}} \times 2^{\mathbb{N} \uplus \overline{\mathbb{N}}}, \subseteq, \cup, (\emptyset, \emptyset))$, where $\subseteq$ and $\cup$ are lifted component-wise. In the following, all uses of t implicitly assert that the label is not $\overline{x}$ for any $x \in \mathbb{N}$.

Lemma 7.1. *For any graph G, $\mathsf{cap}(G)(n, n')$ is the set of keys k for which there exists a path from n to n', every edge of which contains k in its edge label. In particular, if $in(x) = \mathsf{ITE}(x = r, \mathsf{KS}, \emptyset)$ is an inflow on G, then $\mathsf{flow}(in, G)(n)$ is the set of keys k for which such a path exists from r to n.*

If we label each graph edge with its edgeset, then by Lemma 7.1, the flow at each node is the inset of that node. We can encode (GS3) easily using the good condition at each node, and by using sets of keys as the node domain, we can also encode (GS2). We further observe that for graphs with one root, edgesets leaving a node being pairwise disjoint implies that the keysets of every pair of nodes is disjoint[7]. We can thus use a global data structure invariant that the inflow of the root is KS and all other nodes $\emptyset$ to obtain (GS1).

This motivates us to use the following good condition

$$\gamma(x, in, (C, T), f) := \exists t.\ (\gamma_g(x, in, C, t, f) \wedge T = \{t\}\ \vee\ \gamma_b(x, t) \wedge t \neq 0 \wedge T = \{\overline{t}\}) \\ \wedge C \subseteq in(x) \wedge \forall y.\ (C \cap f(x, y) = \emptyset \wedge \forall z.\ f(x, y) \cap f(x, z) = \emptyset)$$

where γ_g and γ_b are user-specified SL predicates. γ_g is to be instantiated with the heap implementation of a node in sync, and γ_b with a description of a node that may not be in sync. The global data structure invariant enforces that the graph has no outgoing edges and that only the root gets a non-zero inflow:

$$\Phi := \exists I, in.\ \mathsf{Gr}(I) \wedge in \in I^{In} \wedge in + \mathbf{0} = \{r \rightarrowtail \mathsf{KS}\} + \mathbf{0} \wedge I^f = \epsilon$$

Lemma 7.2. *The flow graph component of any state satisfying Φ satisfies the good state conditions (GS1) to (GS3).*

The original edgeset paper [Shasha and Goodman 1988] set out three template algorithms, based on different locking disciplines, along with invariants that implied their linearizability. Of these, we formalize the template based on the *give-up* technique here. Our method can be easily extended to the lock-coupling template, but we note that most practical algorithms use more fine-grained locking schemes. For the link technique, the third template, we need to encode the *inreach*, an inductive quantity depending on the keyset and edgesets, viz. k is in the inreach of a node n if k is in the keyset of n or k is in the keyset of n' and for every edge e in the path from n to n', k is in the edgeset of e. Being able to express the inreach would also give a simpler way to prove linearizability of the full Harris list. We would need to extend our framework to support second-order flows to define the inreach, and we leave this for future work.

We now describe the give-up template algorithm, shown in Fig. 9. This template can be used to build implementations of all three dictionary operations by defining the function `decisiveOp` as described below. The basic idea is that every node stores a *range*, which is an under-approximation (i.e. always a subset) of its inset. An operation on key k proceeds by starting at the root and enters a loop at line 3 where it follows edges that contain k in their edgeset. Between two nodes, the algorithm has no locks on either node, to allow for more parallelism. Because of this, when it arrives at some node c, the first thing it does after locking it is to check c's range field (the call to `inRange` in line 6) to ensure that the dictionary operation is at a node whose inset[8] contains k. If the check succeeds, then it calls another helper, `findNext` at line 7, that checks if there exists a node n such that k is in the edgeset of (c, n). If there is no such node, then we know that k must be

[7] For graphs with several roots, each edgeset is defined with respect to each root and then this property holds for each root.
[8] $I^{In}(x)$ is shorthand for $in(x)$ for some $in \in I^{In}$.

in the keyset of c, and we break from the loop (line 8) while holding the lock on c. If the `inRange` check fails, then the algorithm gives up and starts again from the root r (line 13)[9]. If the search continues, the algorithm first unlocks c on line 15 before reassigning c to the next node n.

When the algorithm breaks out of the loop, k must be in the keyset of c (line 15), so it calls `decisiveOp` to perform the operation on c. It then unlocks c, and returns the result of `decisiveOp`.

Proof. The proof annotations for this template (also shown in Fig. 9) are fairly straightforward, and entailments between them can be derived using the lemmas in Fig. 7, assuming that the user-provided implementations of the helper functions satisfy the specifications in Fig. 10. Note that these specifications are in terms of the entire shared state, even though the helper functions only modify the current node c. This is a limitation of RGSep, and one can obtain local specifications for these functions by switching to a more advanced logic such as [Feng 2009]. To reason about interference, we use the following actions to specify the modifications to the shared state allowed by a set of thread IDs T:

$$t \in T \land \mathsf{N}(x, (In, (C, \{0\}), f)) \rightsquigarrow \mathsf{N}(x, (In, (C, T'), f)) \land T' \subseteq \{t, \bar{t}\} \qquad \text{(Lock)}$$

$$t \in T \land emp \rightsquigarrow \mathsf{N}(x, (\{\{x \rightarrowtail 0\}\}, (\emptyset, \{\bar{t}\}), \epsilon)) \qquad \text{(Alloc)}$$

$$t \in T \land \mathsf{Gr}(I) \land I^a \sqsubseteq (_, \{t, \bar{t}\}) \rightsquigarrow \mathsf{Gr}(I') \land I'^a \sqsubseteq (_, \{0, t, \bar{t}\}) \land I \precsim I' \qquad \text{(Sync)}$$

(Lock) allows a thread $t \in T$ to lock an unlocked node; (Alloc) allows t to add new nodes with no inflow, contents, or outgoing edges; and (Sync) allows t to modify a locked region arbitrarily, as long as the new interface contextually extends the old one. The last action also allows it to unlock nodes it has locked. The guarantee of thread with id t_0 is made up of the above actions with $T = \{t_0\}$, while the rely constitutes the above actions with $T = \mathbb{N} \setminus \{t_0\}$. To complete the proof that the template algorithm is memory safe and preserves the global data structure invariant, one must show the stability of every intermediate assertion in Fig. 9. This can be done syntactically using our lemmas, and an example of such a proof can be seen in [Krishna et al. 2017, §C].

Proving Linearizability. To prove that the template algorithm is linearizable, we adapt the Keyset Theorem to the language of flows. To show that all atomic operations maintain the good state conditions, we require that every intermediate assertion about the shared state in our proof implies Φ. This is true of assertions in our template proof, but also needs to be true in the intermediate assertions used to prove that implementations of helper functions meet their specification. The condition about the query key k being in the keyset of the node at which a thread performs its operation is captured by the specifications in Fig. 10. We additionally require that each dictionary operation only modifies the contents of the global once, and that no other operation modifies the contents. We thus obtain the following re-statement of the Keyset Theorem:

Theorem 7.3. *An implementation of the give-up template from Fig. 9 is memory safe and linearizable if the following conditions hold:*

1. *The helper functions satisfy the specification in Fig. 10 under the rely and guarantee specified above, with all intermediate shared-state assertions implying Φ.*
2. *Every execution of* `decisiveOp` *must have at most one call to* **sync** *that changes the contents of the flow graph.*
3. *All other calls to* **sync***, including those in maintenance operations, do not change the contents of the graph region on which they operate.*

[9] We could, in theory, jump to any ancestor of that node. This will require a variable to keep track of the ancestor, and similar reasoning can be used.

Alternatively, our RGSep[FI] proof has established sufficient invariants to directly prove linearizability. One way to do this [Vafeiadis 2009] is to use auxiliary variables to track the abstract state of the data structure (we already have this as the set of contents in the global interface) and atomically execute the specification of each operation on this abstract state at the linearization point. One can then use a write-once variable to store the result of the abstract operation, and at the end of the operation prove that the implementation returns the same value as the specification.

This template algorithm can now be instantiated to any concrete implementation by providing predicates γ_g and γ_b to describe the heap layout of the data structure and implementing the helper functions such that they satisfy the conditions of Theorem 7.3. If there are any maintenance operations, for example splitting or merging nodes, these must also satisfy the invariants in the theorem under the given rely and guarantee. An example implementation of this template, the B+ tree, can be seen in [Krishna et al. 2017, §B].

8 RELATED WORK

Abstraction Mechanisms in Separation Logic. The prevalent mechanism for abstracting unbounded heap regions in separation logic is based on inductive predicates defined by separating conjunctions [Berdine et al. 2004; Brotherston et al. 2011; Cook et al. 2011; Enea et al. 2015; Iosif et al. 2014; Pek et al. 2014; Reynolds 2002]. For simple inductive data structures whose implementation follows regular traversal patterns, inductive predicates can be easier to work with than flow-based abstractions. Certain abstractions, for instance abstracting a list as a sequence of values, while possible to encode in flows, would be more natural with inductive predicates. However, as discussed in §2, inductive predicates are often ill-suited for abstracting concurrent data structures. The reasons include the dependence on specific traversal patterns in the inductive definitions of the predicates, and the implied restrictions on expressing sharing and data structure overlays with separating conjunction.

Ramifications [Hobor and Villard 2013; Mehnert et al. 2012] offer an alternative approach to reasoning about overlaid data structures and data structures with unrestricted sharing. These approaches express inductively defined graph abstractions by using overlapping conjunctions of subheaps instead of separating conjunction. However, this necessitates complicated ramification entailments involving magic wand to reason about updates. We do not need to use combinations of separating conjunction and overlapping conjunction or reason about entailments involving the notorious magic wand. Instead, we shift the reasoning to the composition and decomposition of inflows and flow maps. While showing that a flow graph satisfies conditions on the flow in general requires computing fixpoints over the graph, in proofs we only need to reason about them when a heap region is modified. These computations are typically easy because concurrent algorithms modify a bounded number of nodes at a time in order to minimize interference. Finally, we also obtain a uniform, and decoupled, way to reason about both shape and data properties, yielding abstractions that generalize over a wide variety of data structures such as in our encoding of the edgeset framework.

Yet another alternative for abstracting arbitrary graphs in SL is to use iterated separating conjunction [Müller et al. 2016; Raad et al. 2016; Reynolds 2002; Yang 2001]. Similar to flow interfaces, such abstractions are not tied to specific traversal patterns and can capture invariants that are expressible as local conditions on nodes such as that the graph is closed. However, unlike flow interfaces, iterated separating conjunctions cannot capture inductive properties of a graph (e.g. that the reachable nodes from a root form a tree). In essence, flow interfaces occupy a sweet spot between inductive predicates and iterated separating conjunctions.

Nondeterministic Monoidal Models of Separation Logic. Our notion of flow graph composition naturally yields a nondeterministic monoidal model of SL where separating conjunction $*$ is interpreted as a ternary relation. However, the conventional meta-theory of SL [Calcagno et al. 2007; Dockins et al. 2009] requires $*$ to be partial-deterministic. We overcome this mismatch here by defining an appropriate equivalence relation on flow graphs (or, more precisely, their inflows) to enforce a functional interpretation of $*$. However, this solution leads to a slightly stronger model than is strictly necessary. It also adds some artificial complexity to the logic as the inflow equivalence classes must be reasoned about at the syntactic level. We believe that both of these issues can be avoided by considering a nondeterministic monoidal semantics as the starting point for the development of the program logic. Such semantics have been studied in the context of substructural logics, Boolean BI [Galmiche and Larchey-Wendling 2006], and more recently to obtain a more general proof theory of propositional abstract SL [Hóu et al. 2014]. To our knowledge, flow graphs constitute the first example of a separation algebra with nondeterministic monoidal structure that has practical applications in program verification.

Concurrent Separation Logics. CSL was introduced by O'Hearn [2004]. A recent article by Brookes and O'Hearn [2016] provides a survey of the development of CSLs since then. Among the many improvements that have been developed are the idea of fractional permissions to reason about shared reads [Bornat et al. 2005; Heule et al. 2013], combinations of rely/guarantee reasoning [Feng et al. 2007; Vafeiadis and Parkinson 2007], and abstraction mechanisms for reasoning about different aspects of concurrency such as synchronization protocols for low-level lock implementations [Nanevski et al. 2014], and atomicity abstractions [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Xiong et al. 2017]. The abstractions provided by the latter are orthogonal to the ones developed here. These logics have been used, e.g., to verify specifications of concurrent dictionary implementations based on B-trees and skip lists that enable compositional client verification [da Rocha Pinto et al. 2011; Xiong et al. 2017]. We believe that the proofs developed in [Xiong et al. 2017] can be further simplified by introducing flow interfaces as an intermediate abstraction of the considered data structures.

Higher-order concurrent separation logic [Jung et al. 2015; Krebbers et al. 2017] can express ghost state within the assertion language of the logic itself. This feature can be used to eliminate the restriction of RGSep[FI] that the semantics of the flow interface predicates is defined on the meta level and that it does not support nesting of flow interface abstractions (i.e., cases where a node of a flow graph should abstract from another flow graph contained in the node). Similarly, higher-order CSL can express complex linearizability proofs directly without relegating a part of the proof argument to the meta level.

Invariant Inference. There is a large body of work on inferring invariants for heap-manipulating programs (e.g., [Sagiv et al. 2002]), including techniques based on separation logic [Calcagno et al. 2009; Distefano et al. 2006; Vafeiadis 2010]. Many of these approaches rely on some form of abstract interpretation [Cousot and Cousot 1977]. We believe that the least fixpoint characterization of flows in flow interfaces lends itself well to abstract interpretation techniques.

9 CONCLUSION

We have introduced flow interfaces as a novel approach to the abstraction of unbounded data structures in separation logic. The approach avoids several limitations of common solutions to such abstraction, allows unrestricted sharing and arbitrary traversals of heap regions, and provides a uniform treatment of data constraints. We have shown that flow interfaces are particularly well suited for reasoning about concurrent data structures and that they hold great promise for developing automated techniques for reasoning about implementation-agnostic abstractions.

REFERENCES

Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *FSTTCS*. Springer.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 259–270. https://doi.org/10.1145/1040305.1040327

Richard Bornat, Cristiano Calcagno, and Hongseok Yang. 2006. Variables As Resource in Separation Logic. *Electron. Notes Theor. Comput. Sci.* 155 (May 2006), 247–276. https://doi.org/10.1016/j.entcs.2005.11.059

Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. 2012. Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data. In *ATVA (LNCS)*, Vol. 7561. Springer, 167–182.

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65. https://doi.org/10.1145/2984450.2984457

James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated Cyclic Entailment Proofs in Separation Logic. In *23rd International Conference on Automated Deduction, CADE-23 (Lecture Notes in Computer Science)*, Vol. 6803. Springer, 131–146. https://doi.org/10.1007/978-3-642-22438-6_12

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *POPL*.

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE Computer Society, 366–378. https://doi.org/10.1109/LICS.2007.30

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 234–245. https://doi.org/10.1145/1993498.1993526

Byron Cook, Christoph Haase, Joël Ouaknine, Matthew Parkinson, and James Worrell. 2011. Tractable Reasoning in a Fragment of Separation Logic. In *CONCUR*. Springer.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark J. Wheelhouse. 2011. A simple abstraction for complex concurrent indexes. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*. ACM, 845–864. https://doi.org/10.1145/2048066.2048131

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *28th European Conference on Object-Oriented Programming, ECOOP 2014 (Lecture Notes in Computer Science)*, Vol. 8586. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular Termination Verification for Non-blocking Concurrency. In *25th European Symposium on Programming, ESOP 2016 (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 176–201. https://doi.org/10.1007/978-3-662-49498-1_8

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*. ACM, 287–300. https://doi.org/10.1145/2429069.2429104

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *24th European Conference on Object-Oriented Programming, ECOOP 2010 (Lecture Notes in Computer Science)*, Vol. 6183. Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 (Lecture Notes in Computer Science)*, Vol. 3920. Springer, 287–302. https://doi.org/10.1007/11691372_19

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems, APLAS 2009*. Springer-Verlag, Berlin, Heidelberg, 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. 2016. Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic. *ACM Trans. Program. Lang. Syst.* 38, 2, Article 4 (Jan. 2016), 72 pages. https://doi.org/10.1145/2818638

Constantin Enea, Ondrej Lengál, Mihaela Sighireanu, and Tomás Vojnar. 2017. SPEN: A Solver for Separation Logic. In *NASA Formal Methods, NFM 2017 (Lecture Notes in Computer Science)*, Vol. 10227. Springer, 302–309. https://doi.org/10.1007/978-3-319-57288-8_22

Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. 2015. On Automated Lemma Generation for Separation Logic with Inductive Definitions. In *13th International Symposium on Automated Technology for Verification and Analysis, ATVA 2015 (Lecture Notes in Computer Science)*, Vol. 9364. Springer, 80–96. https://doi.org/10.1007/978-3-319-24953-7_7

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 315–327. https://doi.org/10.1145/1480881.1480922

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *16th European Symposium on Programming, ESOP 2007 (Lecture Notes in Computer Science)*, Vol. 4421. Springer, 173–188. https://doi.org/10.1007/978-3-540-71316-6_13

Didier Galmiche and Dominique Larchey-Wendling. 2006. Expressivity Properties of Boolean BI Through Relational Models. In *26th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2006*, S. Arun-Kumar and Naveen Garg (Eds.). Springer, Berlin, Heidelberg, 357–368. https://doi.org/10.1007/11944836_33

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. ACM, 595–608. https://doi.org/10.1145/2676726.2676975

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings (Lecture Notes in Computer Science)*, Vol. 2180. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013 (Lecture Notes in Computer Science)*, Vol. 7737. Springer, 315–334. https://doi.org/10.1007/978-3-642-35873-9_20

Aquinas Hobor and Jules Villard. 2013. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 523–536. https://doi.org/10.1145/2429069.2429131

Zhé Hóu, Ranald Clouston, Rajeev Goré, and Alwen Tiu. 2014. Proof Search for Propositional Abstract Separation Logics via Labelled Sequents. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*. ACM, New York, NY, USA, 465–476. https://doi.org/10.1145/2535838.2535864

Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. 2014. Deciding Entailments in Inductive Separation Logic with Tree Automata. In *12th International Symposium on Automated Technology for Verification and Analysis, ATVA 2014 (Lecture Notes in Computer Science)*, Vol. 8837. Springer, 201–218. https://doi.org/10.1007/978-3-319-11936-6_15

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *26th European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2017. Go with the Flow: Compositional Abstractions for Concurrent Data Structures (Extended Version). *CoRR* abs/1711.03272 (2017). arXiv:1711.03272 http://arxiv.org/abs/1711.03272

Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670. https://doi.org/10.1145/319628.319663

Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft. 2012. Formalized Verification of Snapshotable Trees: Separation and Sharing. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings (Lecture Notes in Computer Science)*, Vol. 7152. Springer, 179–195. https://doi.org/10.1007/978-3-642-27705-4_15

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification CAV (Lecture Notes in Computer Science)*, Vol. 9779. Springer, 405–425. https://doi.org/10.1007/978-3-319-41528-4_22

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *23rd European Symposium on Programming, ESOP 2014 (Lecture Notes in Computer Science)*, Vol. 8410. Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

Huu Hai Nguyen and Wei-Ngan Chin. 2008. Enhancing Program Verification with Lemmas. In *20th International Conference on Computer Aided Verification, CAV 2008 (Lecture Notes in Computer Science)*, Vol. 5123. Springer, 355–369. https://doi.org/10.1007/978-3-540-70545-1_34

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings (Lecture Notes in Computer Science)*,

Vol. 3170. Springer, 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural proofs for data structure manipulation in C using separation logic. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 440–451. https://doi.org/10.1145/2594291.2594325

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. Separation logic + superposition calculus = heap theorem prover. In *PLDI*. ACM, 556–566.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *CAV (LNCS)*, Vol. 8044. Springer, 773–789.

Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. 2016. Verifying Concurrent Graph Algorithms. In *Asian Symposium on Programming Languages and Systems APLAS (Lecture Notes in Computer Science)*, Vol. 10017. 314–334. https://doi.org/10.1007/978-3-319-47958-3_17

Andrew Reynolds, Radu Iosif, and Cristina Serban. 2017. Reasoning in the Bernays-Schönfinkel-Ramsey Fragment of Separation Logic. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2017 (Lecture Notes in Computer Science)*, Vol. 10145. Springer, 462–482. https://doi.org/10.1007/978-3-319-52234-0_25

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. https://doi.org/10.1145/514188.514190

Dennis E. Shasha and Nathan Goodman. 1988. Concurrent Search Structure Algorithms. *ACM Trans. Database Syst.* 13, 1 (1988), 53–90. https://doi.org/10.1145/42201.42204

Makoto Tatsuta, Quang Loc Le, and Wei-Ngan Chin. 2016. Decision Procedure for Separation Logic with Inductive Definitions and Presburger Arithmetic. In *14th Asian Symposium on Programming Languages and Systems, APLAS 2016 (Lecture Notes in Computer Science)*, Vol. 10017. Springer, 423–443. https://doi.org/10.1007/978-3-319-47958-3_22

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221

Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Neil D. Jones and Markus Müller-Olm (Eds.), Vol. 5403. Springer, 335–348. https://doi.org/10.1007/978-3-540-93900-9_27

Viktor Vafeiadis. 2010. RGSep Action Inference. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Gilles Barthe and Manuel V. Hermenegildo (Eds.), Vol. 5944. Springer, 345–361. https://doi.org/10.1007/978-3-642-11319-2_25

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *18th International Conference on Concurrency Theory, CONCUR 2007 (Lecture Notes in Computer Science)*, Vol. 4703. Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *26th European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 964–990. https://doi.org/10.1007/978-3-662-54434-1_36

Hongseok Yang. 2001. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of the SPACE Workshop*.