Compositional Abstractions for Verifying Concurrent Data Structures

by

Siddharth Krishna

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2019

Professor Thomas Wies

Acknowledgements

This dissertation would, literally (and in the original sense of the word), not exist if not for my advisor Thomas Wies. Without his patience, support, and generous confidence in me I would have quit my PhD before I even started work on this thesis. I could not have hoped for a better advisor, and I would like to thank him for everything. This dissertation also owes its existence to Dennis Shasha, whose talk in our formal methods seminar started off this line of work. I would like to thank Dennis for being a second advisor to me, for his boundless encouragement and enthusiasm, and for being the kind of researcher that I would like to be when I "grow up".

I also want to thank to the rest of my committee members: Mike Dodds, for spending what must have been quite a chunk of time in performing a close reading of my thesis and whose feedback has been invaluable; Michael Emmi, for teaching me the value of good tooling and for an extremely enjoyable internship and collaboration; and Patrick Cousot, for without whose good-humored lambasting in my qualifying exams I may still be struggling with getting machines to learn formal methods.

I am grateful for having the opportunity to collaborate with other excellent researchers and people during my PhD. A chance conversation at a workshop in 2017 with Alex Summers has led to a productive collaboration and great friendship. Alex constantly inspires me to push my results and think of the bigger picture, and has been the driving force behind the more general and elegant framework presented here. I'd like to thank Philippa Gardner and Emanuele D'Osualdo for inviting me to visit them in Imperial and the very interesting and useful discussions on flows. I also want to thank Marc Brockschmidt for two extremely fun internships, and for being my crisis hotline for the last few years. I must thank Danny Tarlow for patiently teaching me machine learning, Paul Gastin for a fun summer exploring pushdown automata, and Anca Muscholl for guiding me through my first summer abroad, my first workshop talk, and my first paper.

I have been very lucky in finding great friends during all my internships and my time in New York. I do not wish to incriminate anyone by publishing their association with me, or risk selective incrimination, so I will not name names. But thank you all for all the fun times!

Finally, I want to thank my family: Anu, for inspiring me to overcome odds, especially physical odds; Thitha, for teaching me to do things properly and be interested in everything; Chutty, for creating new universes and battling childhood boredom by my side; and Karishma, for always being on my team.

This work was in part supported by the National Science Foundation under grants CCF-1815633 and CCF-1618059.

Abstract

Formal verification has had great success in improving the reliability of real-world software, with projects such as ASTREE, CompCert, and Infer showing that rigorous mathematical analysis can handle the scale of today's cyber-infrastructure. However, despite these successes, many core software components are yet to be verified formally. Concurrent data structures are a class of algorithms that are becoming ubiquitous, as software systems seek to make use of the increasingly parallel design of computers and servers. These data structures use sophisticated algorithms to perform fine-grained synchronization between threads, making them notoriously difficult to design correctly, with bugs being found both in actual implementations and in the designs proposed by experts in peer-reviewed publications. The rapid development and deployment of these concurrent algorithms has resulted in a rift between the algorithms that can be verified by the state-of-the-art techniques and those being developed and used today. The goal of this dissertation is to bridge this gap and bring the certified safety of formal verification to the concurrent data structures used in practice.

Permission-based program logics such as separation logic have been established as the standard technique for verifying programs that manipulate complex heap-based data structures. These logics build on so-called *separation algebras*, which allow expressing properties of heap regions such that modifications to a region do not invalidate properties stated about the remainder of the heap. This concept is key to enabling modular reasoning and also extends to concurrency. However, certain data structure idioms prevalent in real-world programs, especially concurrent programs, are notoriously difficult to reason about, even in these advanced logics (e.g., random access into inductively defined structures, data structure overlays). The underlying issue is that while heaps are naturally related to mathematical graphs, many ubiquitous graph properties are non-local in character. Examples of such properties include reachability between nodes, path lengths, acyclicity and other structural invariants, as well as data invariants which combine with these notions. Reasoning modularly about such global graph properties remains a hard problem, since a local modification can have side-effects on a global property that cannot be easily confined to a small region.

This dissertation addresses the question: which separation algebra can be used to prove that a program maintains a global graph property by reasoning only about the local region modified by the program? We propose a general class of global graph properties, that can be expressed in terms of *flows* – fixpoints of algebraic equations over graphs. Flows can encode structural properties of the heap (e.g. the reachable nodes from the root form a tree), data invariants (e.g. sortedness), as well as combinations of both shape and data constraints of overlaid structures in

a uniform manner. We then introduce the notion of a *flow interface*, an abstraction of a region in the heap, which expresses the constraints and guarantees between the region and its context with respect to the flow. Under a suitable notion of composition that preserves the flow values, we show that flow interfaces form the desired separation algebra.

Building on our theory of flows, we develop the *flow framework*, a general proof technique for modular reasoning about global graph properties over program heaps that can be integrated with existing separation logics. We further devise a strategy for automating this technique using SMT-based verification tools. We have implemented this strategy on top of the verification tool Viper and applied it successfully to a variety of challenging benchmarks including 1) algorithms involving general graphs such as Dijkstra's algorithm and a priority inheritance protocol, 2) inductive data structures such as linked lists and B trees, 3) overlaid data structures such as the Harris list and threaded trees, and 4) object-oriented design patterns such as Composite and Subject/Observer. We are not aware of any single other approach that can handle these examples with the same degree of simplicity or automation.

While the flow framework is applicable to any data structure, its features give rise to a new form of modular reasoning for certain concurrent data structures. Concurrent separation logics already apply modularity on multiple levels to simplify correctness proofs, decomposing them according to program structure, program state, and individual threads. Despite these advances, it remains difficult to achieve proof reuse across different data structure implementations. For the large class of concurrent *search structures*, we demonstrate how one can achieve further proof modularity by decoupling the proof of thread safety from the proof of structural integrity.

We base our work on the *template* algorithms of Shasha and Goodman that dictate how threads interact but abstract from the concrete layout of nodes in memory. By using the flow framework of compositional abstractions in the separation logic Iris, we show how to prove correctness of template algorithms, and how to instantiate them to obtain multiple verified implementations. We demonstrate our approach by formalizing three concurrent search structure templates, based on link, give-up, and lock-coupling synchronization, and deriving implementations based on B-trees, hash tables, and linked lists. These case studies represent algorithms used in real-world file systems and databases, which have so far been beyond the capability of automated or mechanized state-of-the-art verification techniques. Our verification is split between the Coq proof assistant and the deductive verification tool GRASShopper in order to demonstrate that our proof technique and framework can be applied both in fully mechanized proof assistants as well as automated program verifiers. In addition, our approach reduces proof complexity and is able to achieve significant proof reuse.

Contents

List of Figures

List of Tables

1 Introduction

For the last 60 years or so, the processing power of computers has been doubling approximately every 2 years. For most of this time, this growth has been backed by the increase in the number of transistors present on integrated circuit chips, a phenomenon commonly known as Moore's Law. This has also been accompanied by a matching increase in clock speed, the speed at which computers perform each step of computation. However, in the 2000s, computer hardware started reaching the physical limits of clock speed, mostly due to overheating and quantum effects. To counter this, manufacturers have turned to parallel architectures where the extra transistors predicted by Moore's law are being used to provide multiple cores on a single chip, enabling multiple computations to be performed in parallel.

Unfortunately, two processors in parallel does not immediately imply a factor of two increase in speed. To make the most of these multicore machines, software needs to be carefully designed to efficiently divide work into threads, sequences of instructions that can be executed in parallel. Well designed parallel algorithms distribute the workload among threads in a way that minimizes the amount of time spent waiting for each other. A standard way to achieve this is to store any shared data in so-called concurrent data structures, algorithms that store and organize data to facilitate efficient access and modification by multiple threads in parallel. These data structures are now core components of critical applications such as drive-by-wire controllers in cars, database algorithms managing financial, healthcare, and government data, and the software-defined-networks of internet service providers. The research community has risen to to meet this need, and has developed concurrent data structure algorithms that are fast, scalable, and able to adapt to changing workloads.

Unfortunately, these algorithms are also among the most difficult software artifacts to develop correctly. Despite being designed and implemented by experts, the sheer complexity and subtlety of the ways in which different threads can interact with one another means that even these experts often fail to anticipate subtle bugs. These can cause the data to be corrupted or the program to misbehave in unexpected ways. For instance, consider the standard textbook on concurrent algorithms, "The Art of Multiprocessor Programming" [Herlihy and Shavit 2008]. Although written by renowned experts who have developed many of the most widely used concurrent data structures, the errata of the book list several severe but subtle errors in the algorithms included in the book's first edition. There have also been many such examples of mistakes in concurrent algorithms in peer-reviewed articles with (pencil-and-paper) mathematical proofs [Burckhardt et al. 2007; Michael and Scott 1995]. It is clear therefore that we desperately need more systematic and dependable techniques to reason about and ensure correctness of these complex algorithms.

Formal verification is a field of research that aims to use mathematical techniques to prove, in a rigorous and machine-checkable manner, the absence of bugs and the conformity of a system to its intended specification. Several projects have demonstrated the successful use of formal verification to improve the reliability of real-world software designs, including SLAM, ASTREE, CertiKOS, seL4, CompCert, and Infer. In fact, in some areas such as hardware verification, formal verification is now a core part of the design process. However, as we argue below, there is a huge gap between the concurrent data structures that can be verified by state-of-the-art techniques and the algorithms being developed and used everyday. The goal of this dissertation is to bridge this gap and bring the certified safety of formal verification to the concurrent data structures in use everyday.

1.1 The Flow Framework

The core technique for formal reasoning about data structures, both sequential and concurrent, is separation logic (SL) [O'Hearn et al. 2001; Reynolds 2002]. SL provides the basis of many successful verification tools that can verify programs manipulating complex data structures [Calcagno et al. 2015; Jacobs et al. 2011; Müller et al. 2016; Appel 2012]. This success is due to the logic's support for reasoning modularly about modifications to heap-based data. 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. For simple inductive data structures such as lists and trees, much of this reasoning can be automated [Berdine et al. 2004; Iosif et al. 2014; Katelaan et al. 2019; Piskac et al. 2013; Enea et al. 2017].

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 that are less regular, consist of multiple overlaid data structures, or provide multiple traversal patterns (e.g. threaded trees). However, these idioms are prevalent in real-world implementations such as the fine-grained concurrent data structures found in operating systems and databases (examples include B-link trees [Lehman and Yao 1981] and non-blocking lists with explicit memory management [Harris 2001]). Solutions to these problems have been proposed [Hobor and Villard 2013] but remain difficult to automate.

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 $\mathrm{lseg}(x, y)$, 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; Cook et al. 2011; Pérez and Rybalchenko 2011; Bouajjani et al. 2012; Iosif et al. 2014; Piskac et al. 2013; Tatsuta et al. 2016; Reynolds et al. 2017;

```
1  method acquire(p: Node, r: Node) {
2    if (r.next == null) {
3      r.next := p
4    } else {
5      p.next := r;
6      update(r, 0, p.curr_prio)
7    }
8  }
9  method update(n: Node, from: Int, to: Int) {
10   if (from == to)
11     return
12   var old_prio := n.curr_prio
13   if (from != 0)
14     n.prios := n.prios \ {from}
15   if (to != 0)
16     n.prios := n.prios ∪ {to}
17   n.curr_prio := max(n.prios)
18   if (n.curr_prio != old_prio && n.next != null) {
19     update(n.next, old_prio, n.curr_prio)
20   }
21 }
```

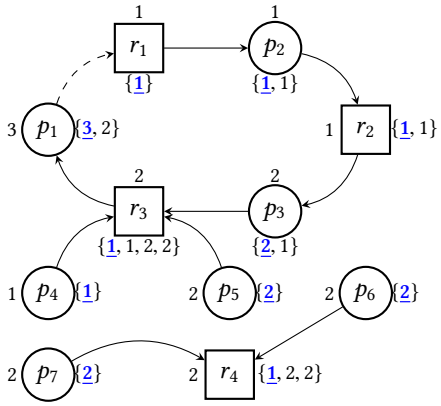


Figure 1.1: Pseudocode of the PIP and a state of the protocol data structure. Round nodes represent processes and rectangular nodes resources. Nodes are marked with their current priorities `curr_prio` as well as the aggregate priority multiset `prios`. A node's default priority is underlined and marked in bold blue.

Enea et al. 2017] or by using heuristics [Nguyen and Chin 2008; Brotherston et al. 2011; Chlipala 2011; Pek et al. 2014; Enea et al. 2015]. However, these techniques can be brittle, in particular, when the predicate definitions involve constraints on data.

For proofs of general graph algorithms, the situation is even more dire. Despite substantial improvements in the verification methodology for such algorithms [Sergey et al. 2015; Raad et al. 2016], significant parts of the proof argument still typically need to be carried out using non-modular reasoning. This dissertation presents a general technique for automated modular reasoning about graph properties that applies to a broad class of data structures and graph algorithms. In fact, for many of the examples that we consider in this dissertation, no fully-modular proof had existed before.

As a motivating example, we consider an idealized version of the priority inheritance protocol (PIP), which is a technique used in process scheduling [Sha et al. 1990]. The purpose of the protocol is to avoid (unbounded) priority inversion, i.e., a situation where a process is blocked from making progress by a lower priority process. The protocol maintains a bipartite graph with nodes representing processes and resources. An example graph is shown in Figure 1.1. An edge from a process p to a resource r indicates that p is waiting for r to become available whereas an edge in the other direction means that r is currently held by p. Every node has an associated default priority as well as a current priority, both of which are strictly positive integers. The current priority affects scheduling decisions. When a process attempts to acquire a resource currently

held by another process, the graph is updated to avoid priority inversion. For example, when process p_1 with current priority 3 attempts to acquire the resource r_1 that is held by process p_2 of priority 2, then p_1's higher priority is propagated to p_2 and, transitively, to any other process that p_2 is waiting for (p_3 in this case). As a result, all nodes on the created cycle will be updated to current priority 3[1]. The protocol thus maintains the following *invariant*: the current priority of each node is the maximum of its default priority and the current priorities of all its predecessors. Priority propagation is implemented by the method `update` shown in Figure 1.1. The implementation represents graph edges by `next` pointers and handles both kinds of modifications to the graph: adding an edge (`acquire`) and removing an edge (`release` - code omitted). To recalculate the current priority of a node (line 17), each node maintains a multiset `prios` which contains the priorities of all its immediate predecessors as well as its own default priority.

Verifying that the PIP maintains its invariant using established separation logic (SL) techniques is challenging. In general, SL assertions describe resources and express the fact that the program has permission to access and manipulate these resources. We stick to the standard model of SL where resources are memory regions represented as partial heaps. Assertions describing larger regions are built from smaller ones using *separating conjunction*, $\phi_1 * \phi_2$. Semantically, the $*$ operator is tied to a notion of resource composition defined by an underlying *separation algebra* [Calcagno et al. 2007; Cao et al. 2017]. In the standard model, composition enforces that ϕ_1 and ϕ_2 must describe disjoint regions. The logic and algebra are set up so that changes to the region ϕ_1 do not affect ϕ_2 (and vice versa). That is, if $\phi_1 * \phi_2$ holds before the modification and ϕ_1 is changed to ϕ_1', then $\phi_1' * \phi_2$ holds afterwards. This so-called *frame rule* enables modular reasoning about modifications to the heap and extends well to the concurrent setting when threads operate on disjoint portions of memory [Brookes and O'Hearn 2016; Dodds et al. 2016; Raad et al. 2015; Dockins et al. 2009]. However, the mere fact that ϕ_2 is preserved by modifications to ϕ_1 does not guarantee that if a global property such as the PIP invariant holds for $\phi_1 * \phi_2$, it also still holds for $\phi_1' * \phi_2$.

For example, consider the PIP scenario depicted in Figure 1.1. If ϕ_1 describes the subgraph containing only node p_1, ϕ_2 the remainder of the graph, and ϕ_1' the graph obtained from ϕ_1 by adding the edge from p_1 to r_1, then the PIP invariant will no longer hold for the new composed graph described by $\phi_1' * \phi_2$. On the other hand, if ϕ_1 captures p_1 and the nodes reachable from r_1 (i.e., the set of nodes modified by `update`), ϕ_2 the remainder of the graph, and we reestablish the PIP invariant locally in ϕ_1 obtaining ϕ_1' (i.e., run `update` to completion), then $\phi_1' * \phi_2$ will also globally satisfy the PIP invariant. The separating conjunction $*$ is not sufficient to differentiate these two cases; both describe valid partitions of a possible program heap. As a consequence, prior techniques have to revert back to non-modular reasoning to prove that the invariant is maintained.

Contributions. In this dissertation we answer the question: which separation algebra allows us to reason modularly about the effects of local changes on global properties of graphs? We consider a general class of global graph properties that can be expressed in terms of *flows* –

[1]The algorithm can then detect the cycle to prevent a deadlock, but we ignore this here to keep the presentation simple.

functions from nodes of the graph to values. For the PIP, the flow maps each node to the multiset of its incoming priorities. One can then use this flow to express the PIP invariant as a node-local condition. In general, a flow is a fixpoint of a set of algebraic equations induced by the graph. These equations are defined over a *flow domain*, which determines how the flow values are propagated along the edges of the graph and how they are aggregated at each node.

In Chapter 3, we present mathematical foundations for flow domains, imposing minimal requirements on the underlying algebra that allow us to capture a broad range of data structure invariants and graph properties, and reason locally about them in a suitable separation algebra. We further identify general mathematical conditions that guarantee unique flows and provide local proof arguments to check the preservation of these conditions.

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. The algebraic properties we require from flow domains 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 enable flow-based proofs in existing separation logic based tools and proof systems, we show that flow interfaces form a separation algebra. Our flow framework can be embedded directly into existing separation logic tools like GRASShopper [Piskac et al. 2014, 2013] and the Iris higher-order separation logic framework [Jung et al. 2015, 2016; Krebbers et al. 2017; Jung et al. 2018a]. The former enables proof automation via SMT solvers and the latter can be used to mechanically check proofs using Coq [Coq Development Team 2017]. This showcases the flexibility of the new framework, as it allows the verifier to choose either highly-trusted but labor-intensive tools or tools that provide more automation but have a larger trusted code base, depending on the needs at hand.

Building on this theory we develop a general proof technique for modular reasoning about global graph properties that can be integrated with existing separation logics (Chapter 4). We further devise a strategy for automating this technique using SMT-based verification tools. We have implemented this strategy on top of the verification tool Viper [Müller et al. 2016] and applied it successfully to a variety of challenging benchmarks. These include 1) algorithms involving general graphs such as the PIP and Dijkstra's algorithm, 2) inductive structures such as linked lists and B trees, 3) overlaid structures such as the Harris list with draining [Harris 2001] and threaded trees, and 4) OO design patterns such as Composite and Subject/Observer. We are not aware of any other approach that can handle these examples with the same degree of simplicity or automation.

1.2 Concurrent Search Structures

An application of our flow framework that exemplifies its advantages is the formalization and verification of concurrent search structure templates. This introduces a new type of modular

proof technique, whereby the proof of thread safety is decoupled from the proof of structural integrity.

Modularity is as important in simplifying formal proofs as it has been for the design and maintenance of large systems. There are three main types of modular proof techniques: (i) Hoare logic [Hoare 1969] enables proofs to be compositional in terms of program structure; (ii) separation logic [O'Hearn et al. 2001; Reynolds 2002] allows proofs of programs to be local in terms of the state they modify; and (iii) thread modular techniques [Jones 1983; Owicki and Gries 1976; Herlihy and Wing 1990] allow one to reason about each thread in isolation.

Concurrent separation logics [O'Hearn 2007; Brookes 2007; Brookes and O'Hearn 2016; Vafeiadis and Parkinson 2007; Feng et al. 2007; Dinsdale-Young et al. 2010, 2013; Fu et al. 2010; Svendsen and Birkedal 2014; Nanevski et al. 2014; da Rocha Pinto et al. 2014; Jung et al. 2015; Dodds et al. 2016] incorporate all of the above techniques and have led to great progress in the verification of practical concurrent data structures, including recent milestones such as a formal proof of the B-link tree [da Rocha Pinto et al. 2011]. Such proofs, however, remain large, complex, and on paper; verified only by hand.

An important reason why existing proofs, such as that of the B-link tree, are still so complicated is that they argue simultaneously about thread safety (i.e., how threads synchronize) and memory safety (i.e., how data is laid out in the heap). We contend that such proofs should instead be decomposed so as to reason about these two aspects independently. When verifying thread safety we should abstract from the concrete heap structure used to represent the data and when verifying memory safety we should abstract from the concrete thread synchronization algorithm. Adding this form of abstraction as a fourth modular proof technique to our arsenal promises more reusable proofs and simpler correctness arguments, which in turn aids proof automation.

As an example, consider the B-link tree, which uses the link-based technique for thread synchronization. The following analogy [Shasha and Goodman 1988] captures the essence of this technique. Bob wants to borrow book k from the library. He looks at the library's catalog to locate k and makes his way to the appropriate shelf n. Before arriving at n, Bob runs into a friend and gets caught up in a conversation. Meanwhile, Alice who works at the library, reorganizes shelf n and moves k as well as some other books to n'. She updates the library catalog and also leaves a sticky note at n indicating the new location of the moved books. Finally, Bob continues his way to n, reads the note, proceeds to n', and takes out k. The synchronization protocol of leaving a note (the *link*) when books are moved ensures that Bob can find k rather than thinking that k is nowhere in the library. However, note that when arguing the correctness of this protocol, we do not need to reason about how books are stored in shelves or how the catalog is organized.

The library patron corresponds to a thread searching for and performing an operation on the key k stored at some node n in the B-link tree and the librarian corresponds to a thread performing a split operation involving nodes n and n'. As in our library analogy, the synchronization technique of creating a forward pointer (the link) when nodes are split is independent of how data is stored within each node and how the nodes are organized in memory (e.g. whether they form a B-tree or a hash table). Hence, it applies to vastly different concrete data structures. Our goal is to verify the correctness of *template algorithms* once and for all so that their proofs can be reused across different data structure implementations.

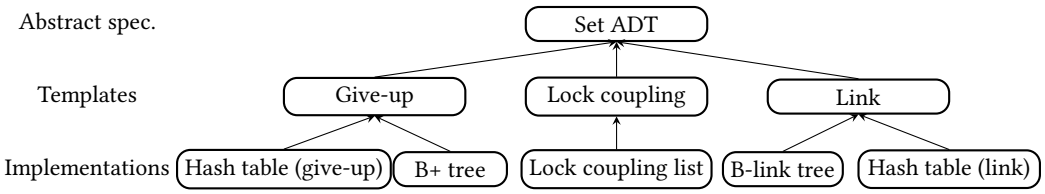


Figure 1.2: The structure of our proofs.

The challenge in achieving this *algorithmic proof modularity* is in reconciling the involved abstractions with the proof technique of reasoning locally about modifications to the heap as in separation logic (SL), which is itself critical for obtaining simple proofs that are easy to mechanize. The proof of the link technique depends on certain invariants about the paths that a search for a key k follows in the data structure graph. However, with the standard heap abstractions used in separation logic (e.g. inductive predicates), it is hard to express these invariants independently of the invariants that capture how the concrete data structure is represented in memory. Consequently, existing proofs such as the one of the B-link tree in [da Rocha Pinto et al. 2011] intertwine the synchronization invariants and the memory invariants, which makes the proof complex, hard to mechanize, and difficult to reuse.

Contributions. In this dissertation we show how to adapt and combine recent advances in separation logic and refinement proofs with our flow-based compositional abstractions in order to achieve the envisioned algorithmic proof modularity for an important class of concurrent data structures: *search structures*.

A search structure is a data structure that supports fast search, insert, and delete operations on a set of key-value pairs (e.g. sorted lists, binary search trees, hash tables, and B-trees). We present a methodology for specifying and verifying template algorithms for concurrent search structures that abstract from the concrete low-level representation of the data structure in memory. The methodology independently verifies that (1) the template algorithm satisfies the abstract specification of search structures assuming that node-level operations maintain certain shape-agnostic invariants and (2) the implementations of these operations for each concrete data structure maintains these invariants. The verification uses SL-style reasoning for both subtasks and the methodology is designed to be usable within off-the-shelf SL-based verification tools so that the proofs can be mechanically checked and automated. Moreover, a key advantage of our approach is that we can perform *sequential* reasoning when we verify that a concrete implementation is a valid instantiation of a template.

We base our work on the template algorithms for concurrent search structures by Shasha and Goodman [1988], who identified the key invariants needed for decoupling reasoning about synchronization and memory representation for such data structures. The second ingredient is our flow framework, which enables us to formalize the correctness argument of Shasha and Goodman [1988] within separation logic.

We demonstrate our methodology by formalizing three template algorithms for concurrent search structures based on the link, the give-up, and the lock-coupling technique of synchronization (Figure 1.2). For these, we derive concrete implementations based on B-trees, hash tables,

and sorted linked lists, resulting in five different data structure implementations. 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.

1.3 Outline

This dissertation begins in Chapter 2 with a survey of the relevant background, and introduces preliminary concepts and notation used throughout. Chapter 3 then presents the flow framework. The general framework is concerned with describing and maintaining properties defined as fixpoints over graphs, and we also present a restricted version of the framework for cases that require the fixpoint to be unique. This chapter is an adapted and generalized version of the flow framework presented at POPL '18 [Krishna et al. 2018a]. Chapter 4 shows how to automate flow-based proofs by presenting a proof strategy that enables SMT-based automation for a variety of difficult data structures. Finally, Chapter 5 uses the flow framework to modularize the proofs of a large class of concurrent search structures. Chapter 6 concludes this dissertation and surveys potential future work.

2 | Preliminaries

We begin with some basic definitions and notations that we use in the rest of this dissertation.

The term $(b\ ?\ t_1 : t_2)$ denotes t_1 if condition b holds and t_2 otherwise. We write $f : A \to B$ for a function from A to B, and $f : A \rightharpoonup B$ for a partial function from A to B. For a partial function f, we write $f(x) = \bot$ if f is undefined at x. We use the lambda notation $(\lambda x.\ E)$ to denote a function that maps x to the expression E (typically containing x). 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 $f[x \rightarrowtail y](z) \coloneqq ((\ ?\ z :=)\ x, y, f(z))$. 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. Given functions $f_1 : A_1 \to B$ and $f_2 : A_2 \to B$ we write $f_1 \uplus f_2$ for the function $f : A_1 \uplus A_2 \to B$ that maps $x \in A_1$ to $f_1(x)$ and $x \in A_2$ to $f_2(x)$ (if A_1 and A_2 are not disjoint sets, $f_1 \uplus f_2$ is undefined).

We write $\delta_{n=n'} : M \to M$ for the function defined by $\delta_{n=n'}(m) \coloneqq m$ if $n = n'$ else 0. We also write $\lambda_0 \coloneqq (\lambda m.\ 0)$ for the identically zero function, $\lambda_{\mathsf{id}} \coloneqq (\lambda m.\ m)$ for the identity function, and use $e \equiv e'$ to denote function equality. For $e : M \to M$ and $m \in M$ we write $m \triangleright e$ to denote the function application $e(m)$. We write $e \circ e'$ to denote function composition, i.e. $(e \circ e')(m) = e(e'(m))$ for $m \in M$, and use superscript notation e^p to denote the function composition of e with itself p times.

For multisets, we use the standard set notation if it is clear from the context. We also write $\{x_1 \rightarrowtail i_1, \ldots, x_n \rightarrowtail i_n\}$ for the multiset containing i_1 occurrences of x_1, i_2 occurrences of x_2, etc. For a multiset S, we write $S(x)$ to denote the number of occurrences of x in S.

Definition 2.1. *A* partial monoid *is a set M, along with a partial binary operation $+ : M \times M \rightharpoonup M$, and a special zero element $0 \in M$, such that (1) $+$ is associative, i.e., $(m_1 + m_2) + m_3 = m_1 + (m_2 + m_3)$; and (2) 0 is an identity element, i.e., $m + 0 = 0 + m = m$. Here, equality means that either both sides are defined and equal, or both sides are undefined.*

We identify a partial monoid with its support set M. If $+$ is a total function, then we call M a monoid. Let $m_1, m_2, m_3 \in M$ be arbitrary elements of the (partial) monoid in the following. We call a (partial) monoid M *commutative* if $+$ is commutative, i.e., $m_1 + m_2 = m_2 + m_1$. Similarly, a commutative monoid M is *cancellative* if $+$ is cancellative, i.e., if $m_1 + m_2 = m_1 + m_3$ is defined, then $m_2 = m_3$.

We say M is *positive* if $m_1 + m_2 = 0$ implies that $m_1 = m_2 = 0$. For a positive monoid M, we can define a partial order $\leq$ on its elements as $m_1 \leq m_2$ if and only if $\exists m_3.\ m_1 + m_3 = m_2$. Positivity also implies that every $m \in M$ satisfies $0 \leq m$.

For $e, e' : M \to M$, we write $e + e'$ for the function that maps $m \in M$ to $e(m) + e'(m)$. We lift this construction to a set of functions E and write it as $\sum_{e \in E} e$.

Definition 2.2. *A function $e : M \to M$ is called an* endomorphism *on M if for every $m_1, m_2 \in M$, $e(m_1 + m_2) = e(m_1) + e(m_2)$. We denote the set of all endomorphisms on M as* $\mathsf{End}(M)$.

Note that for every endomorphism $e \in \mathsf{End}(M)$, $e(0) = 0$ by cancellativity. It is easy to see that $e + e' \in \mathsf{End}(M)$ for any $e, e' \in \mathsf{End}(M)$. Similarly, for $E \subseteq \mathsf{End}(M)$, $\sum_{e \in E} e \in \mathsf{End}(M)$. We say that a set of endomorphisms $E \subseteq \mathsf{End}(M)$ is *closed* if for every $e, e' \in E$, $e \circ e' \in E$ and $e + e' \in E$.

2.1 Separation Logic

Separation logic (SL), is an extension of Hoare logic [Hoare 1969] that is tailored to perform modular reasoning about programs that manipulate mutable resources. The primary application of SL has been to verify heap-based data structures, but the core of SL is an abstract separation logic (based on the logic of bunched implications (BI) [O'Hearn and Pym 1999]) that can be instantiated to obtain various existing forms of SL by choosing an appropriate resource:

Definition 2.3 (Separation Algebra [Calcagno et al. 2007])**.** *A* separation algebra *is a cancellative, partial commutative monoid.*

We will consider two kinds of SL in this dissertation. The first is a standard, first-order separation logic for reasoning about sequential heap-manipulating programs, that we describe in this section. We use this simpler logic in Chapter 4 as a basis to explain how to use the flow framework within SL-based proofs. The second is Iris, a concurrent, higher-order separation logic that can be used to reason about complex fine-grained concurrent programs (§2.2). In Chapter 5, we use the flow framework within Iris to verify complex real-world concurrent search structures. We hope this demonstrates the wide applicability of our flow framework.

We next describe the first-order separation logic (henceforth referred to as "SL").

Heaps Our separation logic uses standard partial heaps as its semantic model. Let us assume we have the following fixed countably infinite sets: Val, consisting of program values; Addr, consisting of memory addresses; and Field, consisting of of field names. Partial heaps are partial maps from addresses to partial maps from field-names to values:

$$\mathsf{Heap} \coloneqq \{h \mid h \colon \mathsf{Addr} \rightharpoonup (\mathsf{Field} \rightharpoonup \mathsf{Val})\}$$

It is easy to see that, under the disjoint union operator $\uplus$, and using the empty heap $h_\emptyset$, $(\mathsf{Heap}, \uplus, h_\emptyset)$ forms a separation algebra.

Programming Language We consider the following simple imperative programming language:

$$
\begin{array}{rcll}
C \in \mathsf{Com} & ::= & \textbf{skip} & \text{Empty command} \\
& | & c & \text{Basic command} \\
& | & C_1;\ C_2 & \text{Sequential composition} \\
& | & C_1 + C_2 & \text{Non-deterministic choice} \\
& | & C^* & \text{Looping} \\
\\
c & ::= & \textbf{assume}(B) & \text{Assume condition} \\
& | & \mathtt{x} := e & \text{Variable assignment} \\
& | & \mathtt{x} := e.f & \text{Heap dereference} \\
& | & e_1.f := e_2 & \text{Heap write} \\
& | & \mathtt{x} := \textbf{alloc}() & \text{Allocate heap cell} \\
& & \ldots &
\end{array}
$$

Here, C stands for commands, c for basic commands, $\mathtt{x}$ for program variables, e for heap-independent expreesions, $f \in \mathsf{Field}$ for field names, and B for boolean expressions. Since we are only concerned with partial correctness in this dissertation, we can define the more familiar program constructs as the following syntactic shorthands:

$$
\begin{aligned}
\mathtt{if}(B)\ C_1\ \mathtt{else}\ C_2 &\coloneqq (\textbf{assume}(B);\ C_1) + (\textbf{assume}(\neg B);\ C_2) \\
\mathtt{while}(B)\ C &\coloneqq (\textbf{assume}(B);\ C)^*;\ \textbf{assume}(\neg B)
\end{aligned}
$$

Assertions We assume that we start from a standard first-order logic over a signature that includes a countably infinite number of uninterpreted functions and predicates. The only requirement on the underlying logic is that it supports additional uninterpreted sorts, functions and predicates, which can be axiomatised in the pure part of the logic[1].

Let Var be an infinite set of variables (we omit sorts and type-checking from the presentation, for simplicity). The syntax of assertions ϕ is given by the following:

$$
\begin{aligned}
\phi ::= {} & P \mid \mathit{true} \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \exists x.\ \phi \\
& \mid e \mapsto \{f_1 : e_1, \ldots\} \mid \phi * \phi \mid \mathop{\Large *}_{x \in X} \phi
\end{aligned}
$$

Here, the first line consists of first order assertions P (called *pure* assertions in the SL world), the always valid assertion *true*, standard boolean connectives, and existential quantification. We can define the remaining boolean connectives and universal quantification as shorthands for the appropriate combination of these. The second line contains the new predicates and connectives introduced by SL (so-called *spatial* assertions). The *points-to* assertion $e \mapsto \{f_1 : e_1, \ldots\}$ is a primitive assertion that denotes a heap cell at adderss e containing fields f_1 with value e_1, etc. The key feature of SL is the new connective $*$, or *separating conjunction*, that is used to conjoin two

[1] We will use this power to express all the values associated with flows and flow interfaces.

disjoint parts of the heap. We use the $\circledast_{x \in X} \phi$ syntax to represent iterated separating conjunction (the bound variable x ranges over a set X)[2].

The semantics of the separation logic assertions are defined with respect to an interpretation of (logical and program) variables $i : \mathrm{Var} \rightharpoonup \mathrm{Val}$. We write $[\![e]\!]_i$ for the denotation of expression e under interpretation i. In particular, we have:

$$h, i \models e \mapsto \{f_1 : e_1, \ldots, f_k : e_k\} \iff h([\![e]\!]_i) = \{f_1 \rightarrowtail e_1, \ldots, f_k \rightarrowtail e_k\}$$
$$h, i \models \phi_1 * \phi_2 \iff \exists h_1, h_2.\ (h = h_1 \uplus h_2) \wedge (h_1, i \models \phi_1) \wedge (h_2, i \models \phi_2)$$

Note that the logic presented here is *garbage-collected* [Cao et al. 2017] (also known as *intuitionistic*). Thus, the semantics of the points-to assertion $x \mapsto \{f_1 : e_1, \ldots, f_k : e_k\}$ does not restrict the heap h to only contain the address x, it only requires x to be included in its domain. This restriction is not essential but simplifies presentation; also, Chapter 4 considers an embedding of the flow framework in the Viper tool, whose logic also uses a garbage-collected semantics.

Operational Semantics We give a small-step operational semantics for our programming language. Configurations are either **fault** or a pair (C, σ) of a command C and a state σ (i.e. a heap-interpretation pair). The following rules define a reduction relation $\rightarrow$ between configurations:

$$\textsc{Seq1}\ \ (\mathbf{skip};\ C_2), \sigma \rightarrow C_2, \sigma \qquad \textsc{Seq2}\ \ \frac{C_1, \sigma \rightarrow C_1', \sigma'}{(C_1;\ C_2), \sigma \rightarrow (C_1';\ C_2), \sigma'} \qquad \textsc{Cho1}\ \ (C_1 + C_2), \sigma \rightarrow C_1, \sigma$$

$$\textsc{Cho2}\ \ (C_1 + C_2), \sigma \rightarrow C_2, \sigma \qquad \textsc{Ass}\ \ \frac{[\![B]\!]_\sigma}{\mathbf{assume}(B), \sigma \rightarrow \mathbf{skip}, \sigma} \qquad \textsc{Loop}\ \ C^*, \sigma \rightarrow (\mathbf{skip} + (C; C^*)), \sigma$$

While we can also give similar small-step semantics to basic commands (for instance, see [Reynolds 2002]), it is easier to understand their axiomatic semantics, presented in the next paragraph.

Soundness of separation logic, especially the frame rule below, relies on the following locality property of the semantics of the programming language. By defining our basic commands via an axiomatic semantics, they automatically satisfy this property, and by construction all composite commands will have the locality property.

Definition 2.4 (Locality). *(L1) If* $(C, \sigma_1 \circledcirc \sigma) \rightarrow^* \mathbf{fault}$, *then* $(C, \sigma_1) \rightarrow^* \mathbf{fault}$.

(L2) If $(C, \sigma_1 \circledcirc \sigma) \rightarrow^* (\mathbf{skip}, \sigma_2)$, *then either there exists* σ_2' *such that* $(C, \sigma_1) \rightarrow^* (\mathbf{skip}, \sigma_2')$ *and* $\sigma_2 = \sigma \circledcirc \sigma_2'$, *or* $(C, \sigma_1) \rightarrow^* \mathbf{fault}$.

[2]Most presentations of SL also include the *separating implication* connective $-\!\!*$. However, logics including $-\!\!*$ are harder to automate and usually undecidable. By omitting $-\!\!*$ we emphasize that we do not require it to perform flow-based reasoning

Proof Rules As with Hoare logic, programs are specified in separation logic by Hoare triples.

Definition 2.5 (Hoare Triple). *We say* $\models \{\phi\}\ C\ \{\psi\}$ *if for every state* σ *such that* $\sigma \models \phi$ *we have (1)* $(C, \sigma) \not\to^* \texttt{fault}$*, and (2) for every state* σ' *such that* $(C, \sigma) \to^* (\texttt{skip}, \sigma')$*,* $\sigma' \models \psi$*.*

In the above definition, $\to^*$ is the reflexive transitive closure of the reduction relation $\to$. Intuitively, the judgment $\{\phi\}\ C\ \{\psi\}$ means that if a command C is executed on a state satisfying the precondition ϕ, then it executes without faults. Moreover, if C terminates, then the resulting state satisfies the postcondition ψ (thus, this is a *partial* correctness criterion).

Separation logic inherits the standard Floyd-Hoare structural proof rules, and the rule of consequence:

$$\text{SL-Skip}\quad \vdash \{\phi\}\ \texttt{skip}\ \{\phi\}$$

$$\text{SL-Seq}\quad \frac{\vdash \{\phi\}\ C_1\ \{\psi\} \qquad \vdash \{\psi\}\ C_2\ \{\rho\}}{\vdash \{\phi\}\ C_1; C_2\ \{\rho\}}$$

$$\text{SL-Choice}\quad \frac{\vdash \{\phi\}\ C_1\ \{\psi\} \qquad \vdash \{\phi\}\ C_2\ \{\psi\}}{\vdash \{\phi\}\ C_1 + C_2\ \{\rho\}}$$

$$\text{SL-Loop}\quad \frac{\vdash \{\phi\}\ C\ \{\phi\}}{\vdash \{\phi\}\ C^*\ \{\phi\}}$$

$$\text{SL-Conseq}\quad \frac{P' \Rightarrow \phi \qquad \vdash \{\phi\}\ C\ \{\psi\} \qquad \psi \Rightarrow Q'}{\vdash \{P'\}\ C\ \{Q'\}}$$

The scalability of SL-based reasoning arises due to the following *frame rule*:

$$\text{SL-Frame}\quad \frac{\vdash \{\phi\}\ C\ \{\psi\}}{\vdash \{\phi * \rho\}\ C\ \{\psi * \rho\}}$$

The frame rule allows one to lift a proof that a command C executes safely on a state satisfying ϕ, producing a state satisfying ψ if it terminates, to the setting where an additional resource ρ (the *frame*) is present. Since C was safe when given only ϕ, it does not access any resources outside ϕ; hence, ρ is untouched in the postcondition. The soundness of the frame rule relies on the disjointness of resources enforced by the separating conjunction operator $*$[3].

For the basic commands of the programming language, one can give *small axioms*, proof rules that specify the minimum resource they need in order to execute safely. The effect of basic commands on more complex states can be derived from these and the frame rule. Here are some of the small axioms:

$$\text{SL-Assign}\quad \vdash \{\psi[x \rightarrowtail e]\}\ \mathtt{x} := e\ \{\psi\} \qquad \text{SL-Write}\quad \vdash \{e_1 \mapsto \{f : \_, \dots\}\}\ e_1.f := e_2\ \{e_1 \mapsto \{f : e_2, \dots\}\}$$

$$\text{SL-Read}\quad \vdash \{e \mapsto \{f : z, \dots\} * e = y\}\ \mathtt{x} := e.f\ \{y \mapsto \{f : z, \dots\} * x = z\}$$

Note that we write $\psi[x \rightarrowtail e]$ for the assertion ψ where all occurrences of x are replaced with e, and _ for an anonymous existential variable (to denote expressions we do not care about).

[3]The frame rule relies on a side condition that the program variables modified by C do not overlap with the free variables in ρ, but this condition can be omitted using the "variables as resource" technique [Bornat et al. 2006].

Together with standard axioms of first-order logic, the proof rules presented above are known to be complete [Yang 2001b][4]. In other words, all valid Hoare triples can be derived by an appropriate combinations of these axioms.

2.2 A Brief Introduction to Iris

Iris is a mechanized higher-order concurrent separation logic framework. A formal introduction to the Iris logic and the underlying programming language semantics is, unfortunately, out of the scope of this dissertation. We provide intuition for the key logical constructs and reasoning steps in this section using a simple example from [Birkedal and Bizjak 2018]; for more details and an excellent introduction to Iris see [Jung et al. 2018b].

Iris's default programming language, and the one we use in this dissertation, is an ML-like language with higher-order store, fork, and compare-and-set (CAS). Again, we do not list the syntax and semantics of the language here, but instead introduce any non-standard features as and when they occur.

Consider the program

$$(\ell \leftarrow !\ell + 1 \mid\mid \ell \leftarrow !\ell + 1)\,;!\ell$$

where $C_1 \mid\mid C_2$ is the parallel composition of two commands (can be encoded using forks). If we know that the value stored at ℓ is initially n, then one specification we might wish to prove is that the program returns a value that is at least $n + 1$. Because we know that the program only increases the value of ℓ, a natural first attempt at an invariant might be $\exists m.\ell \mapsto m \land m \geq n$. However, that is not strong enough to tell us that $!\ell > n$ holds at the end of the program, as it does not rule out decreasing the value at ℓ back to n. Intuitively, what we additionally need to capture is the *protocol* by which threads modify the shared state (in this case, that threads only increase $!\ell$).

One way to model protocols in Iris is to use *ghost state* (also known as logical or auxiliary state, defined formally in §5.2.2), a type of primitive resource analogous to the points-to predicate that helps with the proof but has no effect on run-time behavior. Iris allows us to allocate ghost state at *ghost names*, the analogue of memory addresses for concrete locations. When we allocate a new ghost location, we can pick the type of ghost state stored at that location from any *resource algebra*, a generalization of separation algebras.

In our example, suppose we had a resource algebra with two elements S, a "start token" to encode that $\ell \mapsto n$, and F, a "finish token" to encode that $\exists m.\ \ell \mapsto m \land m > n$. We can allocate a ghost location γ with value S, denoted by the ghost proposition $\boxed{S}^{\gamma}$, at the beginning, and we can update the ghost state to $\boxed{F}^{\gamma}$ when we increase the value at ℓ. Formally, we can tie the value of the ghost location to the value of the physical location using an invariant such as:

$$\mathsf{Inv}_\ell := \exists m.\ \ell \mapsto m * \left(\left(\boxed{S}^{\gamma} \land m = n\right) \lor \left(\boxed{F}^{\gamma} \land m \geq n + 1\right)\right)$$

[4]Note that Yang's completeness result depends crucially on the separating implication $-\!\!*$ being included in the assertion language.

A *resource algebra* is a tuple $(M, \overline{\mathcal{V}} : M \to \mathit{Prop}, |-| : M \to M^?, (\cdot) : M \times M \to M)$ satisfying:

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$
$$\forall a, b.\ a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$
$$\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-CORE-ID)}$$
$$\forall a.\ |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-CORE-IDEM)}$$
$$\forall a, b.\ |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \text{(RA-CORE-MONO)}$$
$$\forall a, b.\ \overline{\mathcal{V}}(a \cdot b) \Rightarrow \overline{\mathcal{V}}(a) \qquad \text{(RA-VALID-OP)}$$

where $M^? := M \uplus \{\bot\}$ $\qquad a^? \cdot \bot := \bot \cdot a^? := a^?$

$a \preccurlyeq b := \exists c \in M.\ b = a \cdot c$

Figure 2.1: The definition of a resource algebra (RA).

This may at first seem pointless: why use ghost state when one can just as easily look at the value stored at ℓ to see whether it is equal to or greater than n?

The advantage of using ghost state is that we can define how ghost state can be shared or combined. For instance, by defining the resource algebra appropriately, we can obtain the following properties of our tokens:

F-DUPLICABLE
$\boxed{F}^\gamma \vdash \boxed{F}^\gamma * \boxed{F}^\gamma$

S-F-INCOMPATIBLE
$\boxed{S}^\gamma * \boxed{F}^\gamma \vdash \mathit{false}$

Once a thread increases the value at ℓ, F-DUPLICABLE lets us duplicate the finish token and share the knowledge that $!\ell > n$. And if we have $\boxed{F}^\gamma$ then together with our invariant S-F-INCOMPATIBLE tells us that $!\ell > n$. We will use this to prove the desired postcondition.

2.2.1 Ghost States and Resource Algebras

Before presenting the proof of our example, we take a small detour and formally define some notions that we will need to perform the above reasoning within Iris.

Resource algebras (RAs) are the structure underlying user-defined resources in Iris[5]. RAs are a generalization of partial commutative monoids or separation algebras, the standard algebraic structure underlying resources in most separation logics (including the one in §2.1).

The definition of a resource algebra is given in Figure 2.1, where *Prop* is the type of propositions of the meta-logic (e.g. Coq). Readers familiar with separation logic will notice that the composition operator is not partial as is the case in standard separation algebras. Instead, RAs use the validity predicate $\overline{\mathcal{V}}$ to identify valid elements of the domain; cases where composition used to be undefined can be encoded by sending them to an invalid element. Another difference

[5]Iris actually uses *cameras* as the structure underlying resources, but as we do not use higher-order resources (i.e. state which can embed propositions) in this dissertation we restrict our attention to RAs, a stronger, but simpler, structure.

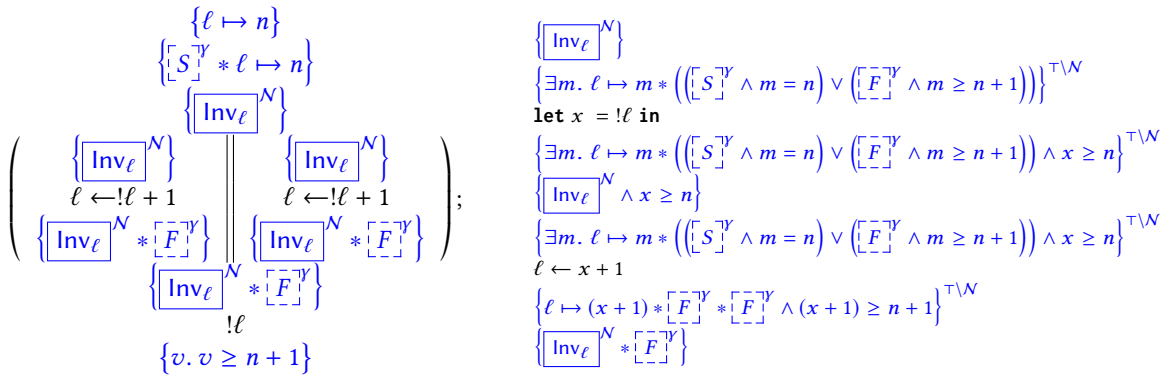


Figure 2.2: Proof sketch of our example program in Iris.

is that RAs do not have a single unit element, and instead the partial function $|-|$ assigns to an element a a *core* $|a|$ (which can be thought of as a's own unit). These features allow Iris to express higher-order state and build more expressive encodings.

For instance, the resource algebra we will use for our example is:

$$M := \{S, F, \lightning\} \qquad \overline{\mathcal{V}}(a) := a \neq \lightning \qquad |F| := F \qquad |S| := \bot \qquad a \cdot b := \begin{cases} F & a = b = F \\ \lightning & \text{otherwise} \end{cases}$$

We can split or compose ghost state at a particular location based on the composition operator of the underlying resource algebra: $\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$. This lets us prove the properties F-DUPLICABLE and S-F-INCOMPATIBLE of our token resource algebra.

The main invariant Iris maintains about ghost locations is that the comopsition of all ghost elements stored at a location is valid as per the underlying resource algebra. For example, Iris allows us to allocate a fresh ghost location containing any valid element. Iris also lets us update the ghost state as long as the invariant that the composite ghost state is valid is maintained.

Definition 2.6. *One can do a* frame-preserving updates *from $a \in M$ to $B \subseteq M$, written $a \rightsquigarrow B$, if*

$$\forall a_{\mathrm{f}}^{?} \in M^{?}.\, \overline{\mathcal{V}}(a \cdot a_f^{?}) \Rightarrow \exists b \in B.\, \overline{\mathcal{V}}(b \cdot a_f^{?}).$$

We say $a \rightsquigarrow b$ if $a \rightsquigarrow \{b\}$.

Intuitively, $a \rightsquigarrow b$ says that every *frame* a_f that is compatible with a should also be compatible with b. Thus, changing one's fragment of the ghost state from a to some b will not violate the assumptions made by anyone else. In our example RA, it is easy to see that the only frame-preserving update is $S \rightsquigarrow F$.

2.2.2 Example Iris Proof

We can now describe the proof of our example program in Iris (shown on the left of Figure 2.2). Since the memory cell located at ℓ needs to be utilized by both threads, we create an *invariant*.

Invariants in Iris, written $\boxed{\phi}^{\mathcal{N}}$ where ϕ is an Iris assertion and $\mathcal{N}$ is the name of the invariant, are *persistent* resources. They are duplicable and thus sharable among multiple threads. To create an invariant $\boxed{\phi}^{\mathcal{N}}$, one must demonstrate that one owns the resource ϕ that it contains.

In our case, we create the invariant $\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}}$, and so we first need to establish Inv_ℓ. We already own the memory location $\ell \mapsto n$, but we do not yet own the ghost resource, so we first allocate a new ghost resource. Iris allows us to allocate a ghost cell $\boxed{a}^{\gamma}$ at a fresh ghost location γ as long as the initial resource is valid, $\overline{\mathcal{V}}(a)$. S is a valid element of our RA, so we allocate the start token $\boxed{S}^{\gamma}$, and now have enough resources (and satisfy the constraints imposed by Inv_ℓ) to create the invariant $\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}}$.

To reason about the parallel composition of threads, Iris has the following (derived) rule[6]:

$$\textsc{HT-Par}\quad \frac{\vdash \{\phi_1\}\ C_1\ \{v.\ \psi_1\} \qquad \vdash \{\phi_2\}\ C_2\ \{v.\ \psi_2\}}{\vdash \{\phi_1 * \phi_2\}\ C_1 \mid\mid C_2\ \{(v_1, v_2).\ \psi_1[v \rightarrowtail v_1] * \psi_2[v \rightarrowtail v_2]\}}$$

This rule allows us to run C_1 and C_2 in parallel if they have *disjoint* preconditions, and breaks down the proof into two separate proofs of each command. Before applying this rule, we use the fact that invariants are duplicable to split $\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}}$ into $\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}} * \boxed{\mathsf{Inv}_\ell}^{\mathcal{N}}$. Now, using HT-Par, we divide up the resources at hand and give each thread one copy of the invariant. Next, we show that each thread satisfies the following Hoare triple: $\left\{\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}}\right\}\ \ell \leftarrow !\ell + 1\ \left\{\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}} * \boxed{F}^{\gamma}\right\}$.

This proof is shown on the right of Figure 2.2, where we have desugared the (non-atomic) read and increment into a read followed by a write. First, to prove that the dereference $!\ell$ in the let binding is safe, we need to exhibit a memory cell $\ell \mapsto \_$. However, at this point we have only the invariant and the finish token, and the memory cell in question is wrapped up in the invariant $\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}}$. Iris allows us to open up invariants and access their contents, but only for an atomic step, and forces us to re-establish them afterwards. We open the invariant $\mathcal{N}$, show that the dereference is safe, and obtain the knowledge that $x \geq n$ (since this is true in either disjunct). Before we can consider the next line of the program, we must close the invariant again (this obligation is denoted by the superscript $\top \setminus \mathcal{N}$ on the intermediate assertion). Since we did not modify the state, we continue to satisfy Inv_ℓ, so we can close up the invariant and obtain $\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}} \wedge x \geq n$.

Now consider the write to ℓ. Again, we need the resource $\ell \mapsto \_$ to show that the write is safe, so we again open the invariant and reason about the write. However, now we have a state where the first disjunct in the invariant no longer holds, so the only way we can close the invariant is to satisfy the second disjunct, which means we need to transform $\boxed{S}^{\gamma}$ into $\boxed{F}^{\gamma}$. Iris allows us to modify ghost state, as long as the change we make to the resource is a frame-preserving update (Definition 2.6). Since we know that $S \rightsquigarrow F$, we convert $\boxed{S}^{\gamma}$ to $\boxed{F}^{\gamma}$.

Finally, to show the desired postcondition of this code fragment, note that we need a copy of

[6]Note that since we are now reasoning about an ML-like programming language, the postcondition in Hoare triples explicitly mention the value v returned by evaluating the command C.

$\boxed{F}^{\gamma}$ outside the invariant. We obtain this by the rule F-DUPLICABLE, and close the invariant (this time, by satisfying the second disjunct).

When the two threads join, HT-PAR lets us compose the postconditions of both threads, obtaining $\boxed{\mathsf{Inv}_\ell}^{\mathcal{N}} * \boxed{F}^{\gamma} * \boxed{\mathsf{Inv}_\ell}^{\mathcal{N}} * \boxed{F}^{\gamma}$. As we are in an intuitionistic setting, we simply throw away the duplicates and obtain the intermediate assertion immediately preceding the final dereference of ℓ in Figure 2.2. Once again, we open the invariant to prove safety of the dereference $!\ell$. Note that this time, since we also own a copy of $\boxed{F}^{\gamma}$ outside the invariant, we have additional information about the shared state. In particular, S-F-INCOMPATIBLE tells us that the first disjunct cannot hold, thus we know that the value stored at ℓ is at least $n + 1$. We thus close the invariant and obtain the postcondition (again throwing away any resources that we no longer need).

3 | The Flow Framework

The flow framework is a separation logic based approach for specifying and reasoning about unbounded data structures. The framework represents the heap as an abstract labeled graph. Data structure invariants are expressed as local conditions satisfied by each node in the graph. These conditions are allowed to depend on the *flow* of the node, a quantity defined as a fixpoint of a set of algebraic equations induced by the entire graph. Unbounded regions of the heap are then abstracted using *flow interfaces* that specify the relies and guarantees that the region imposes on the rest of the heap to maintain the local flow invariants at each of its nodes. Proving that a program preserves the data structure invariants is done by showing that the modified region satisfies an equivalent flow interface.

This chapter begins in §3.1 by explaining the core mathematical notions behind the flow framework. In §3.2 we define the flow as a solution to a fixpoint equation over a graph, and define an abstraction, flow interfaces, to reason compositionally about flow-based properties. In many domains, it is not true that given a graph there exists a flow, or that that the fixpoint equation defining a flow has a unique solution. We describe three conditions in §3.4 under which the flow equation always has a unique solution, and conclude in §3.5. We postpone the surveying of related work to Chapter 4 (see §4.7).

3.1 Flows from First Principles

In this section, we explain the core mathematical notions behind our flow framework, and their motivation with respect to local reasoning principles. We aim for a general technique for modularly proving the preservation of recursively-defined invariants over partial graphs, with well-defined decomposition and composition operations. Partiality is essential for modular reasoning; when applying our reasoning technique to programs, method calls need not be specified with knowledge of the global graph, and in a concurrent setting, multiple threads can simultaneously operate on disjoint subgraphs.

Since we continue to use the priority inheritance protocol as a motivating example through this chapter, we repeat its code and a snapshot of a possible state of the data structure in Figure 3.1.

Flow Domain Our flow framework is parametric with an underlying *flow domain* which is a four-tuple $(M, 0, +, E)$ whose components are elaborated and motivated next. As we will show, different instantiations of these parameters can capture a flexible variety of graph properties

```
1  method acquire(p: Node, r: Node) {
2    if (r.next == null) {
3      r.next := p
4    } else {
5      p.next := r;
6      update(r, 0, p.curr_prio)
7    }
8  }
9  method update(n: Node, from: Int, to: Int) {
10   if (from == to)
11     return
12   var old_prio := n.curr_prio
13   if (from != 0)
14     n.prios := n.prios \ {from}
15   if (to != 0)
16     n.prios := n.prios ∪ {to}
17   n.curr_prio := max(n.prios)
18   if (n.curr_prio != old_prio && n.next != null) {
19     update(n.next, old_prio, n.curr_prio)
20   }
21 }
```

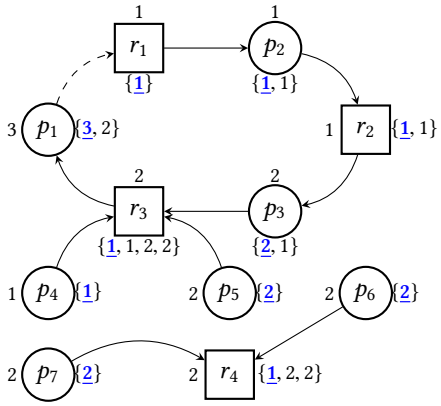


Figure 3.1: Pseudocode of the PIP and a state of the protocol data structure. Round nodes represent processes and rectangular nodes resources. Nodes are marked with their current priorities `curr_prio` as well as the aggregate priority multiset `prios`. A node's default priority is underlined and marked in bold blue.

which can be tracked and reasoned about compatibly with separation-logic-style local reasoning.

Flow Values and Flows General recursive properties of graphs naturally depend on non-local information; for example, we cannot express that a graph is a acyclic directly as a conjunction of independent invariants per node in the graph. To make expressing such properties possible locally, we require a means of summarising this external information, embodied by *flow values* in our technique; the set of flow values M is the first parameter of our flow domains. A flow value is assigned (under constraints explained below) to each node in a graph, capturing sufficient information about the graph to express and reason about non-local properties of interest. Our technique enforces minimal restrictions on the choice of M, which gives it its generality; we consider three examples for the rest of this section:

PIP Domain For reasoning about the priority inheritance protocol (PIP) example (cf. Figure 3.1) flow values capture multisets of integers, representing the priorities of the current node and all those directly referencing it in the PIP data structure.

Path Counting Flow values capture the number of paths from a distinguished root node r; one can then express that a graph is a tree rooted at r with the local condition that the flow at each node is 1.

Inverse Reachability Flow values capture multisets of sets of nodes; each set represents the nodes along a simple path (one with no cycles) leading to the current node in the graph.

For a graph G over a set of nodes N we express properties of G in terms of node-local conditions that may depend on the nodes' *flow*. A flow is a function $\text{flow} : N \to M$ that assigns every node a flow value and must be some fixpoint of the following *flow equation*:

$$\forall n \in N.\ \text{flow}(n) = \mathit{in}(n) + \sum_{n' \in N} \text{flow}(n') \triangleright e(n', n) \tag{3.1}$$

Intuitively, one can think of the flow as being obtained by a fold computation over the graph[1]: the *inflow* $\mathit{in} : N \to M$ defines an initial flow at each node. This initial flow is then updated recursively as follows. For every node n, the current flow value at its predecessor nodes n' is transferred to n via *edge functions* $e(n', n) : M \to M$ (we use $\triangleright$ to denote function application where the function is on the *right*). The transferred flow values are aggregated using the *summation operation* $+$ provided by the flow domain to obtain the updated flow of n. We next motivate the individual components of the flow equation and discuss the constraints imposed on them by local reasoning principles.

Edge Functions In any partial graph, the flow value assigned to a node by a flow is propagated to its neighbours (and transitively) according to a labelling of pairs of nodes (n, n') with *edge functions* $e : M \to M$, mapping the flow value at the *source node* n to one propagated on this edge to the *target node* n'. We require such a labelling for *all* pairs consisting of a source node n inside the graph and a target node n' (possibly outside the graph). In addition, we provide a convenient default case. We require a distinguished 0 flow value to represent no flow (the second element of our flow domains); the corresponding (constant) zero *function* $\lambda_0 = (\lambda m.\ 0)$ then conceptually represents the absence of an edge in the graph[2]. We write $e(n, n')$ for the edge function labelling the pair (n, n'). A set of edge functions E from which this labelling is chosen makes up the fourth element of our flow domains; as we will see in §3.4, restrictions to certain sets E can be exploited to strengthen our overall technique.

For our PIP Domain example, the set of edge functions would be zero functions (where no edge exists in the PIP structure), and otherwise functions which return a singleton multiset storing the maximum value in the input multiset or the default priority of the node, whichever is greater. For instance, in Figure 3.1, $e(r_3, p_2) = \lambda_0$ and $e(r_3, p_1) = (\lambda X.\ \{\max(X \cup \{1\})\})$. Since the flow value at r_3 is $\{1, 2, 2\}$, the edge (r_3, p_1) propagates the value $\{2\}$ to p_1. In the path-counting example, the edge functions would be identity functions (edge present) and zero functions (edge absent). For inverse reachability, the (non-zero) edge function on the edge (n, n') maps a multiset of sets T to a new multiset T' containing $S \cup \{n\}$ for every $S \in T$ such that $n \notin S$. Note, T' does not contain sets $S \in T$ that contain n, because such sets correspond to cyclic paths in the graph, while this particular domain tracks only simple paths.

[1] We note that flows are not generally defined in this manner as we consider any fixpoint of the flow equation to be a flow. Though, the analogy helps to build the right intuition.

[2] We will sometimes informally refer to *paths* in a graph as meaning sequences of nodes for which no edge function labelling a consecutive pair in the sequence is the zero function λ_0.

Flow Aggregation and Inflows The flow value at a node is defined by those propagated to it via edge functions, along with an additional *inflow* value, explained here. Since multiple edges can reach a single node, we need to model the aggregation of these values, for which a binary + operator on flow values must be defined. + is the third element of our flow domains. To make this aggregation of values order-independent, we require + to be commutative and associative. The 0 flow value (representing no flow) must act as a unit with respect to +. For example, in our multiset-based flow domains we let + be multiset union whereas in the path-counting flow domain + means addition on natural numbers.

Every node has an *inflow*, modelling contributions to its flow value which do *not* come from nodes inside the graph. This inflow term plays two important roles: first, since our graphs are partial, the inflow models the contributions from nodes outside of the graph in question. Second, inflow can be artificially added as a means of specialising the computation of flow values. For example, in our PIP domain, the inflow of each node will be the singleton multiset containing the node's default priority. In our path counting domain, we can select the distinguished root node n by giving it an inflow of 1; we could also do this for multiple nodes, to count paths from each. In the inverse reachability domain, we can employ multisets containing a single empty set, forcing paths from these nodes to be tracked by the flow computation.

The flow equation (3.1) defines the flow of a node n to be the aggregation of flow values coming from other nodes n' inside the graph (as given by the respective edge function $e(n', n)$) as well as the inflow $in(n)$. Preserving solutions to this equation is a fundamental goal of our technique. A graph (including its edge functions and flow value assignment) is called a *flow graph* if there is some choice of inflow for its nodes satisfying the flow equation. We now turn to how this property can be preserved under *changes* to the graph in order to aid the verification of flow-dependent invariants.

Graph Updates and Cancellativity Consider that we take a flow graph along with a correct inflow, and obtain a modified graph different only in that a single pair of nodes (n_1, n_2) has a different edge function. We are concerned with the question of whether and how we can change the flow values in the new graph (keeping the inflow unchanged) to satisfy the flow equation.

Consider first the simple case that the target n_2 of the modified edge propagates no flow via edge functions ($e(n_2, n') = \lambda_0$ for all n'); it may however receive additional flow from edge functions $e(n'', n_2)$ coming from nodes n'' other than n_1. For example, in the PIP graph shown in Figure 3.1, removing the edge from p_6 to r_4 (i.e. setting it to the zero function λ_0) does not affect the current priority of r_4 whereas if p_7 had current priority 1 instead of 2, then the current priority of r_4 would have to decrease. For this reason, our PIP Domain aggregates multisets of incoming flow values, rather than having + simply collapse these to their maximum. The multisets contain enough information to locally adjust the flow value when an edge is removed from the graph, whereas if we knew only the maximum and removed an edge (p, r) which provided exactly this value, we could not decide whether or not to decrease the flow value of r without some knowledge of all of r's incoming edges. In this example, recomputing the flow value for r_4 is simply a matter of subtraction (removing $\{2\}$ from the multiset at r_4); this exploits the property that this flow domain is *cancellative* with respect to +, giving us a unique solution. Note that without this

property, the recomputation of a flow value for the target node n_2 of the removed edge consistent with the rest of the graph would in general depend on the values of $\text{flow}(n'') \triangleright e(n'', n_2)$ for *all* nodes n'' (such as p_7 in our example), causing the recomputation to concern unboundedly-many nodes which were not involved in the change to the edge.

Mathematically, cancellativity is the key property which allows removal of edges to be handled without this non-local dependency; for this reason, we make cancellativity of + a *requirement* on our flow domains[3].

Flow Footprints and Interfaces The cancellativity of + alone is not sufficient to reason locally about general flow graph modifications. Consider again the simple modification of changing the edge function labelling a single edge (n_1, n_2). Having cancellativity avoids dependency on arbitrary *incoming* edges, but once we remove the above assumption that the target node n_2 had no non-zero *outgoing* edges, recomputing a single flow value is not sufficient. We also need to account for the propagation of the change transitively throughout the graph. For example, if we add the edge (p_1, r_1) in Figure 3.1 and hence, 3 to the flow of r_1, we also add 3 to the flow of all other nodes reachable from r_1. On the other hand, adding an edge from r_4 to p_5 affects only these two nodes. To capture the relative locality of the side-effects of such updates, we introduce the notion of *flow footprint* of a modification to a flow graph. A modification's flow footprint is the *smallest subset* of the graph containing those nodes which are sources of modified edges, plus all those whose flow values need to be changed in order to obtain a new flow graph. Note that the new flow graph must have the same inflow, as the inflow captures the (unchanged) external contribution to the flow values inside the graph. For example, the flow footprint for the addition of the edge (p_1, r_1) in Figure 3.1 is p_1 and all nodes reachable from r_1 (including r_1 itself).

Flow footprints can be used to localise the effect of a graph modification: from the perspective of the graph *outside* of the flow footprint, nothing observable in the flow domain has changed. We use this observation to extract an *abstraction* of flow graphs which we call *flow interfaces*. Given a flow (sub)graph, its flow interface consists of the node-wise inflow and *outflow* (being the flow contributions its nodes make to all nodes outside of the graph). It is thus an abstraction that hides the flow values and edges inside the region. Flow graphs that have the same flow interface "look the same" to the external graph, as the same values are propagated inwards and outwards.

This idea, while simple, turns out to be powerful enough to build a separation algebra over our reasoning technique, allowing graphs to be decomposed, locally modified and recomposed in ways yielding all the local reasoning benefits of separation logics. In particular, for graph operations within a subgraph with a certain interface, we need to prove: (a) that the modified subgraph is still a flow graph (by checking that the flow equation still has a solution locally in the subgraph) and (b) that it satisfies the same interface (in other words, the flow footprint of the modification is within the subgraph), and the meta-level results for our technique justify that we can recompose the modified subgraph with any graph that the original could be composed with. These steps form the core of our reasoning technique, and are defined formally in §3.2.

[3] As we will show in §3.2.1, an analogous problem for composition of flow graphs is also directly solved by this choice to force aggregation to be cancellative.

Local Reasoning Challenges Our main technique, elaborated in the following section, employs flow interfaces to abstract over subgraphs in which localised graph modifications can be shown to be opaque to the remaining graph. Despite the power of this mechanism, when applying it in practice with a particular flow domain and graphs, the following key questions arise:

1. When does the flow equation *have* a fixpoint solution? How can this be checked, in particular, when is a newly-modified subgraph a flow graph?

2. When is a solution to the flow equation guaranteed to be unique? If desired, how can this property be enforced and preserved?

3. Which graph modifications have which flow footprints? In particular, how localised can their effects on flow values be?

The first two questions are particularly pertinent when we use the flow values of each node in a graph to express properties of interest. For example, consider the path-counting flow domain. If the graph contains a cycle that is reachable from the dedicated root nodes (i.e., those with non-zero inflow), then no flow exists. On the other hand, if the graph contains cycles but no cycle is reachable from the root nodes, then many flows exist; we can assign arbitrary flow values to the cycles as long as the nodes on each cycle are assigned the same value. The property that all nodes have a flow value of 1 expresses that the graph is a tree *only if* we can restrict the fixpoint solution to be one that assigns 0 flow to unreachable cycles.

In the next section, we formalise our base flow framework and core reasoning techniques, and present a variety of techniques for addressing these key questions.

3.2 The Flow Framework

We now present the formal development of our flow framework: a general technique for preserving global properties of graphs using modular reasoning.

3.2.1 Flows and Flow Interfaces

A flow is a recursively-constrained quantity expressed over a labelled graph. The graph labels and flow values are determined by a flow domain; our entire theory is parametric on this domain.

Definition 3.1 (Flow Domain)**.** *A* flow domain $(M, +, 0, E)$ *consists of a commutative cancellative (total) monoid* $(M, 0, +)$ *and a set of functions* $E \subseteq M \to M$*.*

Example 3.2. *The flow domain used for the path-counting flow is* $(\mathbb{N}, +, 0, \{\lambda_{\mathsf{id}}, \lambda_0\})$*, consisting of the monoid on natural numbers under addition and the set of edge functions containing only the identity function and the zero function.*

Example 3.3. *For the PIP, we define the flow domain* $(\mathbb{N}^{\mathbb{N}}, \cup, \emptyset, \{(\lambda S.\ \{\max(S)\}), \lambda_0\})$*, consisting of the monoid of multisets of natural numbers under multiset union and two edge functions:* λ_0 *and the function mapping a multiset* S *to the singleton multiset containing the maximum value in* S*.*

Example 3.4. *Given two flow domains* $(M_1, +_1, 0_1, E_1)$ *and* $(M_2, +_2, 0_2, E_2)$*, the* product *domain* $(M_1 \times M_2, +, (0_1, 0_2), E_1 \times E_2)$ *where* $(m_1, m_2) + (m_1', m_2') := (m_1 +_1 m_1', m_2 +_2 m_2')$ *is a flow domain.*

In the rest of this section we fix a flow domain $(M, +, 0, E)$ and a (potentially infinite) set of nodes $\mathfrak{N}$. We abstract heaps using directed partial graphs; integration of our graph reasoning with direct proofs over program heaps is handled in §4.1. The graphs are partial because they describe abstractions of heaplets rather than the whole heap.

Definition 3.5 (Graph)**.** *A* (partial) graph $G = (N, e)$ *consists of a finite set of nodes* $N \subseteq \mathfrak{N}$ *and a mapping from pairs of nodes to edge functions* $e \colon N \times \mathfrak{N} \to E$*.*

A *flow of graph* $G = (N, e)$ *under inflow* $in \colon N \to M$ is a solution of the following fixpoint equation (the same as (3.1), repeated for clarity) over G, denoted $\mathsf{FlowEqn}(in, e, \mathsf{flow})$:

$$\forall n \in \mathsf{dom}(in).\ \mathsf{flow}(n) = in(n) + \sum_{n' \in \mathsf{dom}(in)} \mathsf{flow}(n') \triangleright e(n', n) \tag{FlowEqn}$$

Example 3.6. *Consider the graph in Figure 3.1; if the flow domain is as in Example 3.3, the inflow function in assigns to every node n the multiset containing n's default priority and we let* $\mathsf{flow}(n)$ *be the multiset labelling every node in the figure, then* $\mathsf{FlowEqn}(in, e, \mathsf{flow})$ *holds.*

An important fact about flows is that any flow of a graph over a product of two flow domains is the product of the flows on each flow domain component; this fact greatly simplifies reasoning about overlaid graph structures. Note that a flow of a graph may (in general) not exist, or may not be unique, depending on the possible solutions to the flow equation (FlowEqn) above.

Definition 3.7 (Flow Graph)**.** *A* flow graph $H = (N, e, \mathsf{flow})$ *consists of a graph* (N, e) *and a function* $\mathsf{flow} \colon N \to M$ *such that there exists an* inflow $in \colon N \to M$ *satisfying* $\mathsf{FlowEqn}(in, e, \mathsf{flow})$*.*

We let $\mathsf{dom}(H) = N$, and sometimes identify H and $\mathsf{dom}(H)$ to ease notational burden. For $n \in H$ we write H_n for the singleton flow subgraph of H induced by n.

Two flow graphs with disjoint domains always compose to a graph, but this will only be a flow graph if their flows are chosen consistently to admit a solution to the resulting flow equation (i.e. the flow graph composition operator defined below is *partial*).

Definition 3.8 (Flow Graph Algebra)**.** *The* flow graph algebra $(\mathsf{FG}, \bullet, H_\emptyset)$ *for the flow domain* $(M, +, 0, E)$ *is defined by*

$$\mathsf{FG} := \{(N, e, \mathsf{flow}) \mid (N, e, \mathsf{flow}) \textit{ is a flow graph}\}$$

$$(N_1, e_1, \mathsf{flow}_1) \bullet (N_2, e_2, \mathsf{flow}_2) := \begin{cases} H & H = (N_1 \uplus N_2, e_1 \uplus e_2, \mathsf{flow}_1 \uplus \mathsf{flow}_2) \in \mathsf{FG} \\ \bot & \textit{otherwise} \end{cases}$$

$$H_\emptyset := (\emptyset, e_\emptyset, \mathsf{flow}_\emptyset)$$

where $e_\emptyset$ *and* $\mathsf{flow}_\emptyset$ *are the edge functions and flow on the empty set of nodes* $N = \emptyset$*. We use* H *to range over* FG*.*

As discussed in §3.1, cancellativity of the flow domain operator + is key to defining an abstraction of flow graphs that permits local reasoning. The following lemma follows from the fact that + is cancellative.

Lemma 3.9. *Given a flow graph* $(N, e, \mathsf{flow}) \in \mathsf{FG}$*, there exists a unique inflow* $in\colon N \to M$ *such that* $\mathsf{FlowEqn}(in, e, \mathsf{flow})$.

Proof. Suppose in and in' are two solutions to $\mathsf{FlowEqn}(\_, e, \mathsf{flow})$. Then, for any n,

$$\mathsf{flow}(n) = in(n) + \sum_{n' \in \mathsf{dom}(in)} \mathsf{flow}(n') \triangleright e(n', n) = in'(n) + \sum_{n' \in \mathsf{dom}(in')} \mathsf{flow}(n') \triangleright e(n', n)$$

which, by cancellativity of the flow domain, implies that $in(n) = in'(n)$. □

Our abstraction of flow graphs consists of two complementary notions. Lemma 3.9 implies that any flow graph has a unique inflow. Thus we can define an inflow function that maps each flow graph $H = (N, e, \mathsf{flow})$ to the unique inflow $\mathsf{inf}(H)\colon H \to M$ such that $\mathsf{FlowEqn}(\mathsf{inf}(H), e, \mathsf{flow})$. We can also define the *outflow* of H as the function $\mathsf{outf}(H)\colon \mathfrak{N} \setminus N \to M$ defined by

$$\mathsf{outf}(H)(n) \coloneqq \sum_{n' \in N} \mathsf{flow}(n') \triangleright e(n', n).$$

Definition 3.10 (Flow Interface)**.** *Given a flow graph* $H \in \mathsf{FG}$*, its* flow interface $\mathsf{int}(H)$ *is the tuple* $(\mathsf{inf}(H), \mathsf{outf}(H))$ *consisting of its inflow and its outflow.*

We use I to range over interfaces, and write I^{in}, I^{out} for the two components of the interface $I = (in, out)$. We again identify I and $\mathsf{dom}(I^{in})$ to ease notational burden. The interfaces of a singleton flow graph containing n capture the flow and the outflow values propagated by n's edges:

Lemma 3.11. *For any flow graph* $H = (N, e, \mathsf{flow})$ *and* $n, n' \in N$*, if* $e(n, n) = \lambda_0$ *then* $\mathsf{int}(H_n)^{in}(n) = \mathsf{flow}(n)$ *and* $\mathsf{int}(H_n)^{out}(n') = \mathsf{flow}(n) \triangleright e(n, n')$.

Proof. Follows directly from (FlowEqn) and the definition of outflow. □

We can now define the *flow interface algebra* as follows:

Definition 3.12 (Flow Interface Algebra)**.**

$$\mathsf{FI} \coloneqq \{\mathsf{int}(H) \mid H \in \mathsf{FG}\}$$
$$I_\emptyset \coloneqq \mathsf{int}(H_\emptyset)$$
$$I_1 \oplus I_2 \coloneqq \begin{cases} I & I_1 \cap I_2 = \emptyset \\ & \quad \wedge \forall i \neq j \in \{1, 2\}, n \in I_i.\ {I_i}^{in}(n) = I^{in}(n) + {I_j}^{out}(n) \\ & \quad \wedge \forall n \notin I.\ I^{out}(n) = {I_1}^{out}(n) + {I_2}^{out}(n) \\ \bot & \textit{otherwise.} \end{cases}$$

Flow interface composition is well defined because of cancellativity of the underlying flow domain (although it is also, like flow graph composition, partial). We next show the key result for this abstraction: the ability for two flow graphs to compose depends only on their interfaces. Flow interfaces thus implicitly define a congruence relation on flow graphs.

Lemma 3.13. $\mathsf{int}(H_1) = I_1 \wedge \mathsf{int}(H_2) = I_2 \Rightarrow \mathsf{int}(H_1 \bullet H_2) = I_1 \oplus I_2$.

Proof. If $H_1 \bullet H_2$ is defined and has interface I, then we show that $I_1 \oplus I_2$ is defined and equal to I. Let $H_i = (N_i, e_i, \mathsf{flow}_i), I = (in, out), I_1 = (in_1, out_1)$, and $I_2 = (in_2, out_2)$. Since $H = H_1 \bullet H_2 \in \mathsf{FG}$ and $\mathsf{inf}(H) = I^{in} = in$, we know by definition that $\forall i \neq j \in \{1, 2\}, n \in H_i$,

$$
\begin{aligned}
& \mathsf{flow}(n) = in(n) + \sum_{n' \in H} \mathsf{flow}(n') \triangleright e(n', n) \\
\iff & \mathsf{flow}_i(n) = in(n) + \sum_{n' \in H} \mathsf{flow}(n') \triangleright e(n', n) \\
\iff & in_i(n) + \sum_{n' \in H_i} e_i(n', n, \mathsf{flow}_i(n')) = in(n) + \sum_{n' \in H_i} e_i(n', n, \mathsf{flow}_i(n')) + \sum_{n' \in H_j} e_j(n', n, \mathsf{flow}_j(n')) \\
\iff & in_i(n) = in(n) + \sum_{n' \in H_j} e_j(n', n, \mathsf{flow}_j(n')) && \text{(By cancellativity)} \\
\iff & in_i(n) = in(n) + out_j(n).
\end{aligned}
$$

Secondly, let $H = H_1 \bullet H_2$ and note that

$$
\begin{aligned}
out(n) &\coloneqq \sum_{n' \in H} \mathsf{flow}(n') \triangleright e(n', n) \\
&= \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n) + \sum_{n' \in H_2} \mathsf{flow}_2(n') \triangleright e_2(n', n) \\
&= out_1(n) + out_2(n).
\end{aligned}
$$

As $H = H_1 \bullet H_2$ implies $\mathsf{dom}(H_1) \cap \mathsf{dom}(H_2) = \emptyset$, this proves that $I_1 \oplus I_2 = I$.

Conversely, if $I_1 \oplus I_2$ is defined and equal to I then we show that $H_1 \bullet H_2$ is defined and has interface I. First, $I_1 \cap I_2 = \emptyset$, so we know that the graphs are disjoint. Note that the proof above works in both directions, so

$$
\begin{aligned}
& \forall i \neq j \in \{1, 2\}, n \in I_i.\ {I_i}^{in}(n) = I^{in}(n) + {I_j}^{out}(n) \\
\Rightarrow\ & \mathsf{flow}(n) = in(n) + \sum_{n' \in H} \mathsf{flow}(n') \triangleright e(n', n).
\end{aligned}
$$

This tells us that $H = H_1 \bullet H_2 \in \mathsf{FG}$ and $\mathsf{inf}(H) = in$. From above, we also know that $out(n) = out_1(n) + out_2(n)$, so the interface composition condition $I^{out}(n) = {I_1}^{out}(n) + {I_2}^{out}(n)$ gives us $\mathsf{outf}(H) = out$. □

Crucially, the following result shows that we can use flow interfaces as an abstraction compatible with separation-logic-style framing. This means that when replacing a flow (sub)graph H_1 with H_1', it is sufficient to check that $\mathsf{int}(H_1')$ is a contextual extension of $\mathsf{int}(H_1)$ in order to ensure that H_1' still composes with anything that H_1 composed with.

Theorem 3.14. *The flow interface algebra* $(\mathsf{FI}, \oplus, I_\emptyset)$ *is a separation algebra.*

We next make the notion of flow footprint that we introduced in §3.1 formally precise.

Definition 3.15 (Flow Footprint)**.** *Let* H *and* H' *be flow graphs such that* $\mathsf{int}(H) = \mathsf{int}(H')$*, then the* flow footprint *of* H *and* H'*, denoted* $\mathsf{ffp}(H, H')$*, is the smallest flow graph* H'_1 *such that there exists* H_1, H_2 *with* $H = H_1 \bullet H_2$*,* $H' = H'_1 \bullet H_2$ *and* $\mathsf{int}(H_1) = \mathsf{int}(H'_1)$*.*

The following lemma states that the flow footprint captures exactly those nodes in the graph that are affected by a modification (i.e. either their flow or their outgoing edges change).

Lemma 3.16. *Let* H *and* H' *be flow graphs such that* $\mathsf{int}(H) = \mathsf{int}(H')$*, then for all* $n \in H$*,* $n \in \mathsf{ffp}(H, H')$ *iff* $H_n \neq H'_n$*.*

In general, when modifying a flow graph H to another flow graph H', requiring that H' satisfies the same interface $\mathsf{int}(H)$ can be too strong a condition. In particular, it does not permit allocating *new nodes* in the modified region. Instead, we want to allow $\mathsf{int}(H')$ to differ from $\mathsf{int}(H)$ in that the new interface could have larger domain, as long as the new nodes are fresh and edges from the new nodes do not change the outflow of the modified region. We capture this notion formally below.

Definition 3.17. *An interface* $I = (in, out)$ *is* contextually extended *by* $I' = (in', out')$*, written* $I \precsim I'$*, if and only if*

1. $\mathsf{dom}(in) \subseteq \mathsf{dom}(in')$*,*
2. $\forall n \in \mathsf{dom}(in).\ in(n) = in'(n)$*, and*
3. $\forall n' \notin \mathsf{dom}(in).\ out(n') = out'(n')$*.*

The following theorem states that contextual extension preserves composability and is itself preserved under interface composition.

Theorem 3.18 (Replacement Theorem)**.** *If* $I = I_1 \oplus I_2$*, and* $I_1 \precsim I'_1$ *are all valid interfaces such that* $I'_1 \cap I_2 = \emptyset$ *and* $\forall n \in I'_1 \setminus I_1.\ {I_2}^{out}(n) = 0$*, then there exists a valid* $I' = I'_1 \oplus I_2$ *such that* $I \precsim I'$*.*

Proof. Let H, H_1, H_2 be flow graphs with interfaces I, I_1, I_2 respectively (thus, $H = H_1 \bullet H_2$). Define $H_0 = (N_0, e_0, \mathsf{flow}_0) \coloneqq (\mathsf{dom}(I'_1) \setminus \mathsf{dom}(I_1), (\lambda n, n'.\ \lambda_0), (\lambda n.\ in'_1(n) + out_1(n)))$.

First, we show that $H'_1 = H_0 \bullet H_1$ is defined. The graph composition is defined because $I'_1 \cap I_2 = \emptyset$, and to show that it is a flow graph, we choose $in'_1 = {I'_1}^{in}$. We need to show, for any $n \in H'_1$, that

$$\begin{aligned}\mathsf{flow}'_1(n) &= in'_1(n) + \sum_{n' \in H'_1} \mathsf{flow}'_1(n') \triangleright e'_1(n', n) \\ &= in'_1(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n). \qquad (\text{as } n' \in H_0 \Rightarrow e'_1(n', n) = \lambda_0)\end{aligned}$$

When $n \in H_0$, by definition,

$$\mathsf{flow}'_1(n) = \mathsf{flow}_0(n) = in'_1(n) + out_1(n) = in'_1(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n).$$

On the other hand, when $n \in H_1$,

$$\mathsf{flow}'_1(n) = \mathsf{flow}_1(n) = in_1(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n) = in'_1(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n)$$

($I_1 \precsim I'_1$ implies $in_1(n) = in'_1(n)$).

Second, we show that $\mathsf{int}(H'_1) = I'_1$. By construction, $\mathsf{inf}(H'_1) = in'_1 = {I'_1}^{in}$. The outflow $\mathsf{outf}(H'_1)(n)$ is equal to

$$\sum_{n' \in H'_1} \mathsf{flow}'_1(n') \triangleright e'_1(n', n) = \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n) = out_1(n) = out'_1(n)$$

(the last equality follows from $I_1 \precsim I'_1$).

Third, we show that $H' = H'_1 \bullet H_2 = H_0 \bullet H$ is a flow graph. The graph composition is defined because $I'_1 \cap I_2 = \emptyset$, and to show that it is a flow graph, we choose the inflow

$$in'(n) = \begin{cases} in(n) & n \in H \\ in'_1(n) & n \in H_0. \end{cases}$$

We now need to show that (FlowEqn) holds, but first note that for any $n \in H'$,

$$\sum_{n' \in H'} \mathsf{flow}'(n') \triangleright e'(n', n) = \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n) + \sum_{n' \in H_2} \mathsf{flow}_2(n') \triangleright e_2(n', n) \tag{3.2}$$

as when $n' \in H_0$, $e'(n', n) = e_0(n', n) = \lambda_0$. Now when $n \in H_0$,

$$\mathsf{flow}'(n) = \mathsf{flow}_0(n) = in'_1(n) + out_1(n) = in'(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n).$$

The RHS of (FlowEqn) is, by (3.2),

$$in'(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n) + \sum_{n' \in H_2} \mathsf{flow}_2(n') \triangleright e_2(n', n)$$

but note that the third term is zero as $n \in I'_1 \setminus I_1 \Rightarrow {I_2}^{out}(n) = 0$. Thus, the flow equation holds. On the other hand, when $n \in H$,

$$\mathsf{flow}'(n) = \mathsf{flow}(n) = in(n) + \sum_{n' \in H} \mathsf{flow}(n') \triangleright e(n', n) = in'(n) + \sum_{n' \in H'} \mathsf{flow}'(n') \triangleright e'(n', n),$$

where the last equality follows from (3.2).

Finally, we let $I' = \mathsf{int}(H')$ and show that $I \precsim I'$. By construction, $\mathsf{dom}(I) \subseteq \mathsf{dom}(I')$ and $in(n) = in'(n)$ when $n \in \mathsf{dom}(I)$. When $n' \notin \mathsf{dom}(I)$, by (3.2), $out'(n') = out_1(n') + out_2(n') = out(n')$, completing the proof. □

3.2.2 Flow Interfaces as a Resource Algebra

As defined above, flow graphs (Definition 3.8) and flow interfaces (Definition 3.12) have a partial composition operator. However, to use flow interfaces to reason about programs within the Iris logic, we need them to be resource algebras (Figure 2.1), in particular composition needs to be a total operator. This can be achieved by defining a special *invalid* element and mapping all pairs for whom composition should be undefined to this invalid element. This subsection presents modified definitions of the flow graph and flow interface algebras that can be used in Iris. Since this is also the standard way of encoding partial functions in first-order logic, we also use this in our SL encoding in Chapter 4.

Definition 3.19 (Flow Graph Algebra)**.** *The* flow graph algebra $(\mathsf{FG}, \bullet, H_\emptyset)$ *for flow domain* $(M, +, 0, E)$ *is defined by*

$$\mathsf{FG} ::= H \in \{(N, e, \mathsf{flow}) \mid (N, e, \mathsf{flow}) \textit{ is a flow graph}\} \mid H_\lightning$$

$$(N_1, e_1, \mathsf{flow}_1) \bullet (N_2, e_2, \mathsf{flow}_2) := \begin{cases} H & H = (N_1 \uplus N_2, e_1 \uplus e_2, \mathsf{flow}_1 \uplus \mathsf{flow}_2) \in \mathsf{FG} \\ H_\lightning & \textit{otherwise} \end{cases}$$

$$\_ \bullet H_\lightning := H_\lightning =: H_\lightning \bullet \_$$

$$H_\emptyset := (e_\emptyset, \mathsf{flow}_\emptyset)$$

where $e_\emptyset$ *and* $\mathsf{flow}_\emptyset$ *are the edge functions and flow on the empty set of nodes* $N = \emptyset$*.*

Definition 3.20 (Flow Interface Algebra)**.** *The flow interface algebra* $(\mathsf{FI}, \overline{\mathcal{V}}, |-|, \oplus)$ *is defined by*

$$\mathsf{FI} ::= I \in \{\mathsf{int}(H) \mid H \in \mathsf{FG}\} \mid I_\lightning \qquad \overline{\mathcal{V}}(a) := a \neq I_\lightning \qquad |a| := I_\emptyset$$

$$I_1 \oplus I_2 := \begin{cases} \mathsf{int}(H) & \exists H_1, H_2.\ H = H_1 \bullet H_2 \wedge \forall i \in \{1, 2\}\ .\ \mathsf{int}(H_i) = I_i \\ I_\lightning & \textit{otherwise,} \end{cases}$$

where $I_\emptyset := \mathsf{int}(H_\emptyset)$*.*

Theorem 3.21. *The flow interface algebra* $(\mathsf{FI}, \overline{\mathcal{V}}, |-|, \oplus)$ *is a resource algebra.*

3.3 Expressivity of Flows

We now give a few examples of flows to demonstrate the range of data structures whose properties can be expressed as local constraints on each node's flow.

We start by demonstrating the generality of the flow equation in terms of its ability to capture global graph properties. In particular, we define a *universal* flow that computes, at each node, sufficient information to reconstruct the entire graph. This shows that flows are powerful enough to capture any graph property of interest.

Definition 3.22 (Universal Flow). *Say we are given a set of nodes $N \subseteq \mathfrak{N}$ and a function $\epsilon : N \times \mathfrak{N} \to A$ labelling each pair of nodes from some set A (for instance, to encode an unlabelled graph, $A = \{0, 1\}$ and $\epsilon(n, n')$ is 1 iff an edge is present in the graph). Consider the flow domain $(\mathbb{N}^{2^{\mathfrak{N}\times\mathfrak{N}\times A}}, \cup, \emptyset, E)$, consisting of the monoid of multisets of sets of tuples (n, n', a) of edges (n, n') and labels $a \in A$ under multiset union and edge functions E containing λ_0 and for every $n, n' \in \mathfrak{N}$, $a \in A$ the function*

$$\lambda_{n,n',a}(S) \coloneqq \{P \rightarrowtail ((n, n', a) \in P \ ? \ S(P \setminus \{(n, n', a)\}) : 0)\} .$$

Given a flow graph $H = (N, e, \mathsf{flow})$, if $e(n, n') = \lambda_{n,n',\epsilon(n,n')}$ and $\mathsf{inf}(H) = (\lambda n.\ \{\emptyset\})$, then $\mathsf{flow}(n)$ is a multiset containing, for each simple path in H ending at n, a set of all edge-label tuples of edges occurring on that path.

To see why the universal flow computes the entire graph at each node, let us look at the edge functions in more detail. The way to think of $e(n, n') = \lambda_{n,n',\epsilon(n,n')}$ is that it looks at each path P' in the input multiset S and if P' does not contain the tuple $(n, n', \epsilon(n, n'))$ then it adds the tuple to P' and adds the resulting path P to the output multiset. In order to convert this procedure into a multiset comprehension style definition, the formal definition above starts from each path P in the output multiset and works backward (i.e. $P' = P \setminus \{(n, n', \epsilon(n, n'))\}$).

To understand the flow computation, let us start with the inflow to a node n, the singleton multiset containing the empty set $\{\emptyset\}$, and track its progress through a path. For every n', the edge function $e(n, n')$ acts on $\{\emptyset\}$ and propagates the singleton multiset $\{(n, n', \epsilon(n, n'))\}$. In this way, if we consider a sequence of (distinct) edges $(n_1, n_2), \ldots, (n_{k-1}, n_k)$, then this value becomes $\{(n_1, n_2, \epsilon(n_1, n_2)), \ldots, (n_{k-1}, n_k, \epsilon(n_{k-1}, n_k))\}$. However, the minute we follow an edge (n_i, n_{i+1}) that has occurred on the path before, the edge function $e(n_i, n_{i+1})$ will send this value to the empty multiset $\emptyset$. Thus, the flow at each node n turns out to be the multiset containing sets of edge-label tuples for each simple path in the graph[4]. Note that we label all pairs of nodes in the graph by edges of the form $\lambda_{n,n',\epsilon(n,n')}$. This means that $\mathsf{flow}(n)$ will contain one set for every sequence of pairs of nodes in the graph, even those corresponding to edges that do not "exist" in the original graph ϵ. From this information, one can easily reconstruct all of ϵ and hence any graph property of the global graph.

The power of the universal flow to capture any graph property comes with a cost: the flow footprint of any modification is the entire global graph. This means that we lose all powers of local reasoning, and revert to expensive global reasoning about the program. This is to be expected, however, because the universal flow captures all details of the graph, even ones that are possibly irrelevant to the correctness of the program at hand. The art of using flows is to carefully define a flow that captures exactly the necessary global information needed to locally prove correctness of a given program. The rest of this section describes a few interesting examples of flows that we use in our case studies in Chapter 4.

Definition 3.23 (Path-counting Flow). *The path-counting flow uses the flow domain $(\mathbb{N}, +, 0, \{\lambda_{\mathsf{id}}, \lambda_0\})$ defined in Example 3.2. Given a flow graph $H = (N, e, \mathsf{flow})$ over this domain, if $e(n, n')$ is λ_{id} for edges that are present in the graph and λ_0 otherwise, and $\mathsf{inf}(H) = (\lambda n.\ (n = r \ ? \ 1 : 0))$ for some root node r, then $\mathsf{flow}(n)$ is the number of paths from r to n.*

[4]This flow domain has the property that any graph has a unique solution to the flow equation (see §3.4.2).

The path-counting flow is a very useful flow for describing the shape of common structures, e.g. lists (singly and doubly linked, cyclic), trees, and (by using product flow constructions) nested and overlaid 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.

Definition 3.24 (Inverse Reachability Flow)**.** *Consider the flow domain* $(\mathbb{N}^{2^{\mathfrak{N}}}, \cup, \emptyset, E)$*, consisting of the monoid of multisets of sets of nodes under multiset union and edge functions* E *containing* λ_0 *and for every* $n \in \mathfrak{N}$ *the function*

$$\lambda_n(S) \coloneqq \{P \rightarrowtail (n \in P \ ? \ S(P \setminus \{n\}) : 0)\} .$$

Given a flow graph $H = (N, e, \mathsf{flow})$*, if* $e(n, n') = \lambda_n$ *and* $\mathsf{inf}(H) = (\lambda n.\ \ (n = r \ ? \ \{\emptyset\} : \emptyset))$*, then* $\mathsf{flow}(n)$ *is a multiset containing, for each simple path in* H *from* r *to* n*, the set of all nodes occurring on that path.*

This flow is a simplified version of the universal flow in that for each edge it only keeps track of the source node. By capturing less information about the global graph, however, this flow permits more modifications: for instance, one can swap the order of two nodes in a simple path and only update the flows of the two nodes modified. This is an example of carefully tuning the flow domain to match the modifications performed by the program.

3.4 Existence and Uniqueness of Flows

We typically express global properties of a graph $G = (N, e)$ by fixing a global inflow $in : N \to M$ and then constraining the flow of each node in N using node-local conditions. However, as we discussed at the end of §3.1, there is no general guarantee that a flow exists or is unique for a given in and G. The remainder of this section presents three complementary conditions under which we can prove that our flow fixpoint equation always has a unique solution. To this end, we say that a flow domain $(M, +, 0, E)$ has *unique flows* if for every graph (N, e) over this flow domain and inflow $in\colon N \to M$, there exists a unique flow that satisfies the flow equation $\mathsf{FlowEqn}(in, e, \mathsf{flow})$.

3.4.1 Edge-local Flows

A simple but useful case is when all edge functions $e \in E$ *ignore* their input; i.e. are constant functions. We call such a flow domain *edge-local*. In this case, the flow of every node can be computed as a direct aggregation (according to the flow domain operator $+$) of the (constant) values its neighbours edge functions propagate; the flow equation is no-longer recursive and always has a unique solution.

Example 3.25. *The flow of a PIP graph can be encoded using an edge-local flow domain. This is because the PIP implementation tracks the flow explicitly as part of the state of each object (the multisets stored in the* `prios` *field). We explain this in more depth in §4.3.*

Lemma 3.26. *If $(M, +, 0, E)$ is a flow domain such that for every $e \in E$ there exists $a \in M$ such that $e \equiv (\lambda m.\ a)$, then this flow domain has unique flows.*

Proof. Let (N, e) be a graph, $in\colon N \to M$ an inflow, and $\mathsf{flow}\colon N \to M$ an arbitrary flow that satisfies $\mathsf{FlowEqn}(in, e, \mathsf{flow})$. We will give a closed expression for $\mathsf{flow}(n)$, thereby showing that there is a unique flow.

Since the flow domain is edge-local, there must be a constant $a_{n,n'}$ such that for every $n, n' \in N$, $e(n, n') \equiv (\lambda m.\ a_{n,n'})$. By (FlowEqn), for any $n \in N$,

$$\mathsf{flow}(n) = in(n) + \sum_{n' \in N} \mathsf{flow}(n') \triangleright e(n', n) = in(n) + \sum_{n' \in N} a_{n',n}.$$

As this is a closed expression depending only on the graph and the inflow, the flow must be unique. □

3.4.2 Nilpotent Cycles

Let $(M, +, 0, E)$ be a flow domain where every edge function $e \in E$ is an endomorphism on M. In this case, we can show that the flow of a node n is the sum of the flow as computed along *each path* in the graph that ends at n. Suppose we additionally know that the edge functions are defined such that their composition along any *cycle* in the graph eventually becomes the identically zero function. In this case, we need only consider finitely many paths to compute the flow of a node, which means the flow equation has a unique solution.

Formally, such edge functions are called *nilpotent endomorphisms*:

Definition 3.27. *A closed set of endomorphisms $E \subseteq \mathsf{End}(M)$ is called* nilpotent *if there exists $p > 1$ such that $e^p \equiv 0$ for every $e \in E$.*

Example 3.28. *The edge functions of the inverse reachability domain of §3.1 are nilpotent endomorphisms (taking $p = 2$).*

If all edges of a flow graph are labelled with edges from a nilpotent set of endomorphisms, then the flow equation has a unique solution:

Lemma 3.29. *If $(M, +, 0, E)$ is a flow domain such that M is a positive monoid and E is a nilpotent set of endomorphisms, then this flow domain has unique flows.*

Before we prove this lemma, we present some useful notions and lemmas when dealing with flow domains that are endomorphisms.

Lemma 3.30. *If $(M, +, 0, E)$ is a flow domain such that E is a closed set of endomorphisms, $G = (N, e)$ is a graph, $in\colon N \to M$ is an inflow such that $\mathsf{FlowEqn}(in, e, \mathsf{flow})$, and $L \geq 1$,*

$$\begin{aligned}\mathsf{flow}(n) = in(n) &+ \sum_{\substack{n_1,\ldots,n_k \in N \\ 1 \leq k < L}} in(n_1) \triangleright e(n_1, n_2) \cdots e(n_{k-1}, n_k) \triangleright e(n_k, n) \\ &+ \sum_{n_1,\ldots,n_L \in N} \mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{L-1}, n_L) \triangleright e(n_L, n).\end{aligned}$$

Proof. If $L = 1$, then this follows directly from (FlowEqn): $\mathsf{flow}(n) = \mathit{in}(n) + \sum_{n' \in N} \mathsf{flow}(n') \triangleright e(n', n)$. For $L > 1$, by induction, we can assume that

$$\mathsf{flow}(n) = \mathit{in}(n) + \sum_{\substack{n_1,\ldots,n_k \in N \\ 1 \le k < L-1}} \mathit{in}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{k-1}, n_k) \triangleright e(n_k, n) + \sum_{n_1,\ldots,n_{L-1} \in N} \mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{L-2}, n_{L-1}) \triangleright e(n_{L-1}, n).$$

Now, using the (FlowEqn) to substitute for $\mathsf{flow}(n_1)$ in the third term, we get (after using the endomorphism property and some variable renaming):

$$\sum_{n_1,\ldots,n_{L-1} \in N} \mathit{in}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{L-2}, n_{L-1}) \triangleright e(n_{L-1}, n) + \sum_{n_1,\ldots,n_L \in N} \mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{L-1}, n_L) \triangleright e(n_L, n).$$

Adding these to the above expression completes the proof. □

Definition 3.31. *The* capacity *of a flow graph $G = (N, e)$ is* $\mathsf{cap}(G) \colon N \times \mathfrak{N} \to (M \to M)$ *defined inductively as* $\mathsf{cap}(G) \coloneqq \mathsf{cap}^{|G|}(G)$*, where*

$$\mathsf{cap}^0(G)(n, n') \coloneqq \delta_{n=n'} \qquad \mathsf{cap}^{i+1}(G)(n, n') \coloneqq \delta_{n=n'} + \sum_{n'' \in G} \mathsf{cap}^i(G)(n, n'') \circ e(n'', n').$$

For a flow graph $H = (N, e, \mathsf{flow})$, we write $\mathsf{cap}(H)(n, n') = \mathsf{cap}((N, e))(n, n')$ for the capacity of the underlying graph. Intuitively, $\mathsf{cap}(G)(n, n')$ is the function that summarizes how flow is routed from any source node n in G to any other node n', including those outside of G.

Lemma 3.32. *The capacity is equal to the following sum-of-paths expression:*

$$\mathsf{cap}^i(G)(n, n') = \delta_{n=n'} + \sum_{\substack{n_1,\ldots,n_k \in G \\ 0 \le k < i}} e(n, n_1) \cdots e(n_k, n').$$

Proof. Follows from the definition of capacity by a straightforward induction on i. □

We now have all the ingredients needed to prove that nilpotent flow domains have unique flows.

Proof of Lemma 3.29. Let $G = (N, e)$ be a graph, $\mathit{in} \colon N \to M$ an inflow, and $\mathsf{flow} \colon N \to M$ an arbitrary flow that satisfies $\mathsf{FlowEqn}(\mathit{in}, e, \mathsf{flow})$. By Lemma 3.30,

$$\mathsf{flow}(n) = \mathit{in}(n) + \sum_{\substack{n_1,\ldots,n_k \in N \\ 1 \le k < L}} \mathit{in}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{k-1}, n_k) \triangleright e(n_k, n) + \sum_{n_1,\ldots,n_L \in N} \mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{L-1}, n_L) \triangleright e(n_L, n). \tag{3.3}$$

Let S be the third term in (3.3); we claim S is zero for sufficiently large L. When $L > p|N|$, where p is the degree of nilpotence of E, we will show that (in terms of the monoid order $\leq$):

$$S \leq \sum_{n_1, m \in N} \mathsf{flow}(n_1) \triangleright \mathsf{cap}^L(G)(n_1, m) \triangleright \left(\mathsf{cap}^L(G)(m, m)\right)^p \triangleright \mathsf{cap}^L(G)(m, n).$$

In this case, since E is nilpotent, $\left(\mathsf{cap}^L(G)(m, m)\right)^p \equiv \lambda_0$ which implies that $S \leq 0 \implies S = 0$. To show the above inequality, consider any term $\mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{L-1}, n_L) \triangleright e(n_L, n)$ in S. As $L > p|N|$, by the Pigeonhole Principle there must be some $m \in N$ that appears $p + 1$ times in the sequence $n_1, \ldots, n_L$. Thus, there exists some indices $1 \leq i_1 < \cdots < i_{p+1} \leq L$ such that our term is equal to

$$\begin{aligned}
&\mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{i_1 - 1}, m) \\
&\qquad \triangleright \underbrace{\left(e(m, n_{i_1+1}) \cdots e(n_{i_2-1}, m)\right) \cdots \left(e(m, n_{i_p+1}) \cdots e(n_{i_{p+1}-1}, m)\right)}_{p \text{ times}} \\
&\qquad \triangleright e(m, n_{i_{p+1}+1}) \cdots e(n_{L-1}, n_L) \triangleright e(n_L, n).
\end{aligned}$$

By Lemma 3.32, this is $\leq \mathsf{flow}(n_1) \triangleright \mathsf{cap}^L(G)(n_1, m) \triangleright \left(\mathsf{cap}^L(G)(m, m)\right)^p \triangleright \mathsf{cap}^L(G)(m, n)$.

Thus,

$$\mathsf{flow}(n) = \mathit{in}(n) + \sum_{\substack{n_1, \ldots, n_k \in N \\ 1 \leq k < L}} \mathit{in}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{k-1}, n_k) \triangleright e(n_k, n).$$

As this expression is uniquely determined by the flow domain, the inflow, and the graph, the flow must be unique. □

3.4.3 Effectively Acyclic Flow Graphs

There are some flow domains that compute flows useful in practice, but which do not guarantee either existence or uniqueness of fixpoints *a priori* for all graphs. For example, the path-counting flow from Example 3.2 is one where for certain graphs, there exist no solutions to the flow equation (see Figure 3.2(a)), and for others, there can exist more than one (in Figure 3.2(b), the nodes marked with x can have any path count, as long as they both have the same value).

In such cases, we explore how to restrict the class of *graphs* we use in our flow-based proofs such that each graph has a unique fixpoint; the difficulty is that this restriction must be respected for composition of our graphs. Here, we study the class of flow domains $(M, +, 0, E)$ such that M is a positive monoid and E is a set of *reduced* endomorphisms (defined below); in such domains we can decompose the flow computations into the various paths in the graph, and achieve unique fixpoints by restricting the kinds of cycles graphs can have.

Definition 3.33. *A flow graph* $H = (N, e, \mathsf{flow})$ *is* effectively acyclic (EA) *if for every* $1 \leq k$ *and* $n_1, \ldots, n_k \in N$,

$$\mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{k-1}, n_k) \triangleright e(n_k, n_1) = 0.$$

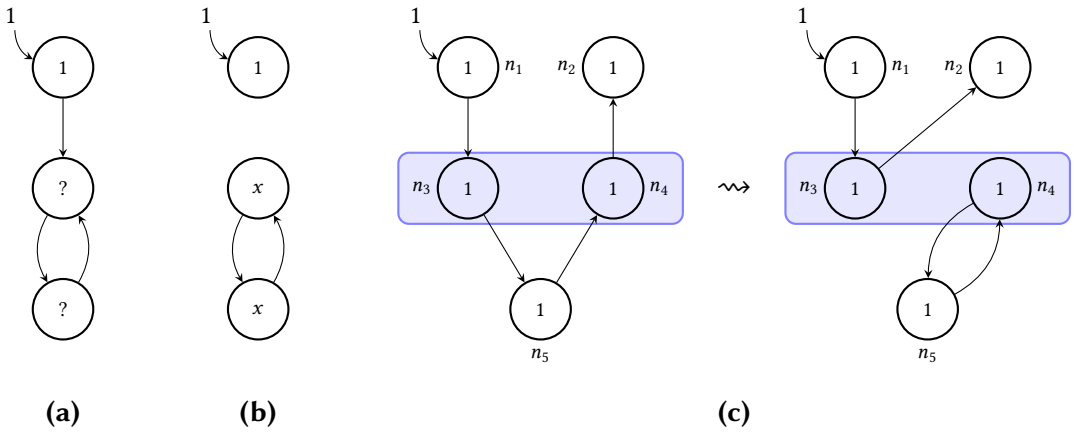


Figure 3.2: Examples of graphs that motivate effective acyclicity. All graphs use the path-counting flow domain, the flow is displayed inside each node, and the inflow is displayed as curved arrows to the top-left of nodes. 3.2(a) shows a graph and inflow that has no solution to (FlowEqn); 3.2(b) has many solutions. 3.2(c) shows a modification that preserves the interface of the modified nodes, yet goes from a graph that has a unique flow to one that has no solutions to (FlowEqn).

The simplest example of an effectively acyclic graph is one where the edges with non-zero edge functions form an acyclic graph. However, our semantic condition is weaker: for example, when reasoning about two overlaid acyclic lists whose union happens to form a cycle, a product of two path-counting domains will satisfy effective acyclicity because the composition of different types of edges results in the zero function.

Lemma 3.34. *Let $(M, +, 0, E)$ be a flow domain such that M is a positive monoid and E is a closed set of endomorphisms. Given a graph (N, e) over this flow domain and inflow $in\colon N \to M$, if there exists a flow graph $H = (N, e, \mathsf{flow})$ that is effectively acyclic, then* flow *is unique.*

Proof. By Lemma 3.30, if we can show that for some $L > 1$ the following expression is 0, then the flow at every node is determined uniquely by the inflow and the graph:

$$\sum_{n_1,\ldots,n_L \in N} \mathsf{flow}(n_1) \triangleright e(n_1, n_2) \cdots e(n_{L-1}, n_L) \triangleright e(n_L, n). \tag{3.4}$$

Pick $L > |G|$. For any term T in (3.4), the Pigeonhole Principle tells us there must be some $m \in N$ that appears twice in the sequence $n_1, \ldots, n_L$. Thus, there exists indices $1 \le i_1 < i_2 \le L$ such that

$$T = \mathsf{flow}(n_1) \triangleright \underbrace{e(n_1, n_2) \cdots e(n_{i_1-1}, m)}_{T_1} \triangleright \underbrace{e(m, n_{i_1+1}) \cdots e(n_{i_2-1}, m)}_{T_2} \triangleright \underbrace{e(m, n_{i_2+1}) \cdots e(n_{L-1}, n_L) \triangleright e(n_L, n)}_{T_3}.$$

Note that by Lemma 3.30, $\mathsf{flow}(n_1) \triangleright T_1 \leq \mathsf{flow}(m)$. Furthermore, as H is EA, $\mathsf{flow}(m) \triangleright T_2 = 0$. Thus, $T = \mathsf{flow}(n_1) \triangleright T_1 \triangleright T_2 \triangleright T_3 \leq \mathsf{flow}(m) \triangleright T_2 \triangleright T_3 = 0 \triangleright T_3 = 0$. By positivity, $T = 0$, completing the proof. □

While the restriction to effectively acyclic flow graphs guarantees us that the flow is the unique fixpoint of the flow equation, it is not easy to show that modifications to the graph preserve EA while reasoning locally. Even modifying a subgraph to another with the same flow interface (which we know guarantees that it will compose with any context) can inadvertently create a cycle in the larger composite graph. For instance, consider Figure 3.2(c), that shows a modification to nodes $\{n_3, n_4\}$ (the boxed blue region). The interface of this region is $(\{n_3 \rightarrowtail 1, n_4 \rightarrowtail 1\}, \{n_5 \rightarrowtail 1, n_2 \rightarrowtail 1\})$, and so swapping the edges of n_3 and n_4 preserves this interface. However, the resulting graph, despite composing with the context to form a valid flow graph, is not EA (in this case, it has multiple solutions to the flow equation). This shows that flow interfaces are not powerful enough to preserve effective acyclicity. For a special class of endomorphisms, we show that a local property of the modified subgraph can be checked, which implies that the modified composite graph continues to be EA.

Definition 3.35. *A closed set of endomorphisms $E \subseteq \mathsf{End}(M)$ is called* reduced *if $e \circ e \equiv \lambda_0$ implies $e \equiv \lambda_0$ for every $e \in E$.*

Note that if E is reduced, then no $e \in E$ can be nilpotent. In that sense, this class of instantiations is complementary to those in §3.4.2.

Example 3.36. *Examples of flow domains that fall into this class include positive semirings of reduced rings (with the additive monoid of the semiring being the aggregation monoid of the flow domain and E being any set of functions that multiply their argument with a constant flow value). Note that any direct product of integral rings is a reduced ring. Hence, products of the path counting flow domain are a special case.*

For reduced endomorphisms, it is sufficient to check that a modification preserves the flow routed between every pair of source and sink node. This pairwise check ensures that we do not create any new cycles in any larger graph. We define a relation analogous to contextual extension, that constrains us to modifications that preserve EA while allowing us to allocate new nodes[5].

Definition 3.37. *A flow graph H' is a* subflow-preserving extension *of H, written $H \precsim_s H'$, if*

$$\begin{aligned} \mathsf{int}(H) &\precsim \mathsf{int}(H'), \\ \forall n \in H, n' \notin H', m.\ m \leq \mathsf{inf}(H)(n) &\Rightarrow m \triangleright \mathsf{cap}(H)(n, n') = m \triangleright \mathsf{cap}(H')(n, n'), \text{ and} \\ \forall n \in H' \setminus H, n' \notin H', m.\ m \leq \mathsf{inf}(H')(n) &\Rightarrow m \triangleright \mathsf{cap}(H')(n, n') = 0. \end{aligned}$$

We now show that it is sufficient to check our local condition on a modified subgraph to guarantee composition back to an effectively-acyclic composite graph:

[5]The monoid ordering used in the following definition exists because we are working with a positive monoid (see Chapter 2).

Theorem 3.38. *Let $(M, +, 0, E)$ be a flow domain such that M is a positive monoid and E is a reduced set of endomorphisms. If $H = H_1 \bullet H_2$ and $H_1 \lesssim_s H_1'$ are all effectively acyclic flow graphs such that $H_1' \cap H_2 = \emptyset$ and $\forall n \in H_1' \setminus H_1.\ \mathsf{outf}(H_2)(n) = 0$, then there exists an effectively acyclic flow graph $H' = H_1' \bullet H_2$ such that $H \lesssim_s H'$.*

Before we prove this Theorem, we list some properties of flow graph compositions where one subgraph consists of disconnected nodes (all edges are zero function):

Lemma 3.39. *If $H' = H_0 \bullet H$ such that $H_0 = (\_, e_0, \_)$ and $\forall n, n'.\ e_0(n, n') \equiv \lambda_0$, and H is EA, then H' is EA.*

Lemma 3.40. *If $H' = H_0 \bullet H$ such that $H_0 = (\_, e_0, \_)$ and $\forall n, n'.\ e_0(n, n') \equiv \lambda_0$, then $\forall n' \notin H'.\ \mathsf{outf}(H')(n') = \mathsf{outf}(H)(n')$.*

Lemma 3.41. *If $H' = H_0 \bullet H$ such that $H_0 = (\_, e_0, \_)$ and $\forall n, n'.\ e_0(n, n') \equiv \lambda_0$, then*

$$\forall n \in H', n' \notin H', m.\ m \leq \mathsf{inf}(H')(n) \Rightarrow m \triangleright \mathsf{cap}(H')(n, n') = \begin{cases} m \triangleright \mathsf{cap}(H)(n, n') & n \in H \\ 0 & n \in H_0. \end{cases} \tag{3.5}$$

We now have enough tools to prove the analogue of the Replacement Theorem for effectively acyclic graphs.

Proof of Theorem 3.38. We first consider the case where $\mathsf{dom}(H_1) = \mathsf{dom}(H_1')$. In this case, note that the definitions of contextual and subflow-preserving extensions reduce to $\mathsf{int}(H_1) = \mathsf{int}(H_1')$ and $\forall n \in H_1, n' \notin H_1$,

$$m \leq \mathsf{inf}(H_1)(n) \Rightarrow m \triangleright \mathsf{cap}(H_1)(n, n') = m \triangleright \mathsf{cap}(H_1')(n, n'). \tag{3.6}$$

Thus, Lemma 3.13 tells us that $H' = H_1' \bullet H_2$ exists and $\mathsf{int}(H') = \mathsf{int}(H)$. All that remains to show is that H' is EA and show the analogue of (3.6) between H and H'. To simplify the proof, let $H_2' \coloneqq H_2$ and note that (3.6) also holds between H_2 and H_2'.

Let us first show that H' is EA. If not, then there exists some $n_1^1, n_1^2, \ldots, n_1^{p_1}, n_2^1, \ldots, n_k^{p_k} \in H$ that is an alternating sequence (i.e. n_i^1 and n_{i+1}^1 are in different subgraphs, and every n_i^j is in the same subgraph as n_i^1) and some $1 \leq x \leq k, 1 \leq y \leq p_x$ such that

$$S = in(n_1^1) \triangleright e'(n_1^1, n_1^2) \cdots e'(n_k^{p_k - 1}, n_k^{p_k}) \triangleright e'(n_k^{p_k}, n_x^y) \neq 0.$$

Note that k cannot be 1, because then the entire path would be in H_i' for $i \in \{1, 2\}$, contradicting the fact that H_i' is EA; thus $1 < k$.

If $x = k$, then let

$$T = in(n_1^1) \triangleright \left(\sum_{\substack{m_1, \ldots, m_l \in H_- \\ 0 \leq l < L}} e'(n_1^1, m_1) \cdots e'(m_l, n_2^1) \right) \cdots \left(\sum_{\substack{m_1, \ldots, m_l \in H_- \\ 0 \leq l < L}} e'(n_{k-1}^1, m_1) \cdots e'(m_l, n_k^1) \right)$$

for some L such that $S + \_ = T \triangleright e'(n_k^1, n_k^2) \cdots e'(n_k^{p_k}, n_x^y)$. As $S \neq 0$, positivity tells us $S + \_ \neq 0$, and so $T \neq 0$. By applying (3.6) repeatedly, we can show that $T = in(n_1^1) \triangleright c\_(n_1^1, n_2^1) \ldots c\_(n_{k-1}^1, n_k^1)$. The definition of interface composition tells us that $in\_(n_k^1) = T + \_$, so by positivity we must have $in\_(n_k^1) \triangleright e'(n_k^1, n_k^2) \cdots e'(n_k^{p_k}, n_x^y) \neq 0$. However, this is a path entirely in some subgraph, contradicting EA of that subgraph.

On the other hand, if $x < k$, we have

$$S = in(n_1^1) \triangleright e'(n_1^1, n_1^2) \cdots e'(n_x^{y-1}, n_x^y) \triangleright \left(e'(n_x^y, n_x^{y+1}) \cdots e'(n_x^{p_x}, n_{x+1}^1)\right) \triangleright e'(n_{x+1}^1, n_{x+1}^2) \cdots e'(n_k^{p_k}, n_x^y).$$

Since our flow domain is reduced and $S \neq 0$, we can repeat the parenthesized portion and obtain

$$\begin{aligned} S' = in(n_1^1) \triangleright e'(n_1^1, n_1^2) \cdots e'(n_x^{y-1}, n_x^y) \triangleright e'(n_x^y, n_x^{y+1}) \cdots e'(n_x^{p_x}, n_{x+1}^1) \\ \triangleright e'(n_{x+1}^1, n_{x+1}^2) \cdots e'(n_k^{p_k}, n_x^y) \triangleright e'(n_x^y, n_x^{y+1}) \cdots e'(n_x^{p_x}, n_{x+1}^1) \neq 0. \end{aligned}$$

Note that this is a path that starts and ends on the boundary (i.e. with the last edge crossing between subgraphs). Depending on whether n_x^y and n_k^1 are in the same subgraph or not, we can use new variable names and write this as

$$S' = in(m_1^1) \triangleright e'(m_1^1, m_1^2) \cdots e'(m_l^{q_l - 1}, m_l^{q_l}) \triangleright e'(m_l^{q_l}, m_{l+1}^1) \neq 0.$$

Let

$$T = in(m_1^1) \triangleright \left(\sum_{\substack{n_1, \ldots, n_k \in H\_ \\ 0 \leq k < L}} e'(m_1^1, n_1) \cdots e'(n_k, m_2^1) \right) \cdots \left(\sum_{\substack{n_1, \ldots, n_k \in H\_ \\ 0 \leq k < L}} e'(m_l^1, n_1) \cdots e'(m_k, n_{l+1}^1) \right)$$

for some L such that $T = S' + \_$, so by positivity, $T \neq 0$. Again, applying (3.6) repeatedly lets us show that $T = in(m_1^1) \triangleright c\_(m_1^1, m_2^1) \ldots c\_(m_l^1, m_{l+1}^1)$. By expanding the definitions of the capacities and using the endomorphism property we get that T is the sum of cyclic paths in H, at least one of which is non-zero. This contradicts the fact that H is EA. The proof of the analogue of (3.6) between H and H' proceeds similarly, by breaking up all paths into segments in each subgraph and using (3.6) repeatedly; we omit it here for clarity of presentation.

We now turn to the case where $\mathrm{dom}(H_1) \subset \mathrm{dom}(H_1')$. Our aim is to reduce this case to the previous case, thus we need to construct an intermediate graph H_1'' that satisfies the conditions of this theorem but additionally has the same domain as H_1'.

We define $H_0 = (N_0, e_0, \mathrm{flow}_0) \coloneqq (\mathrm{dom}(H_1') \setminus \mathrm{dom}(H_1), (\lambda n, n'.\ \lambda_0), (\lambda n.\ in_1'(n) + out_1(n)))$. Let $H_1'' = H_0 \bullet H_1$; this exists because the graphs are disjoint by construction, and we show that (FlowEqn) is true for inflow in_1' by showing, for any $n \in H_1''$, that

$$\begin{aligned} \mathrm{flow}_1''(n) &= in_1'(n) + \sum_{n' \in H_1''} \mathrm{flow}_1''(n') \triangleright e_1''(n', n) \\ &= in_1'(n) + \sum_{n' \in H_1} \mathrm{flow}_1(n') \triangleright e_1(n', n). && (\text{as } n' \in H_0 \implies e_1''(n', n) = \lambda_0) \end{aligned}$$

When $n \in H_0$, by definition,

$$\mathsf{flow}''_1(n) = \mathsf{flow}_0(n) = \mathit{in}'_1(n) + \mathit{out}_1(n) = \mathit{in}'_1(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n).$$

On the other hand, when $n \in H_1$,

$$\mathsf{flow}''_1(n) = \mathsf{flow}_1(n) = \mathit{in}_1(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n) = \mathit{in}'_1(n) + \sum_{n' \in H_1} \mathsf{flow}_1(n') \triangleright e_1(n', n),$$

where we use $\mathit{in}_1(n) = \mathit{in}'_1(n)$ from $H_1 \precsim_s H'_1$. By Lemma 3.39, we get that H''_1 is EA.

Next, we show that $H''_1 \precsim_s H'_1$. First, to show that $I''_1 \precsim I'_1$, note that by construction $\mathsf{dom}(H''_1) = \mathsf{dom}(H'_1)$; we have already shown that H''_1 satisfies the flow equation with inflow in'_1, hence they have the same inflows; and by Lemma 3.40 they have the same outflows. Second, let $n \in H''_1, n' \notin H'_1$ and $m \leq \mathsf{inf}(H''_1)(n)$. If $n \in H_0$, then by Lemma 3.41, $m \triangleright \mathsf{cap}(H''_1)(n, n') = 0$ and since $n \notin H_1$, by $H_1 \precsim_s H'_1$, $m \triangleright \mathsf{cap}(H'_1)(n, n') = 0$. On the other hand, if $n \in H_1$, then by Lemma 3.41, $m \triangleright \mathsf{cap}(H''_1)(n, n') = m \triangleright \mathsf{cap}(H_1)(n, n')$ which equals $m \triangleright \mathsf{cap}(H'_1)(n, n')$ by $H_1 \precsim_s H'_1$ and $\mathsf{inf}(H''_1)(n) = \mathsf{inf}(H'_1)(n) = \mathsf{inf}(H_1)(n)$. The third condition of $H''_1 \precsim_s H'_1$ is vacuously true since $\mathsf{dom}(H''_1) = \mathsf{dom}(H'_1)$.

By a similar argument, we can show that $H'' = H''_1 \bullet H_2 = H_0 \bullet H$ exists, and is EA. At this point, we have $H'' = H''_1 \bullet H_2$ and $H''_1 \precsim_s H'_1$ are all effectively acyclic flow graphs such that $H'_1 \cap H_2 = \emptyset$ and additionally $\mathsf{dom}(H''_1) = \mathsf{dom}(H'_1)$. Thus, we can use the case when the domains are equal (which has been proved above) and obtain an EA flow graph $H' = H'_1 \bullet H_2$ such that $H'' \precsim_s H'$. Since $\precsim_s$ is transitive, we just need to show that $H \precsim_s H''$ to complete the proof.

To see that $H \precsim_s H''$, first consider $I \precsim I''$: by construction and $H_1 \precsim_s H'_1$, $\mathsf{dom}(H) \subseteq \mathsf{dom}(H'')$; we constructed H'' to have an inflow that is equal to in on all nodes in H; and by Lemma 3.40 they have the same outflows. Second, let $n \in H, n' \notin H''$ and $m \leq \mathsf{inf}(H)(n)$. By Lemma 3.41, $m \triangleright \mathsf{cap}(H'')(n, n') = m \triangleright \mathsf{cap}(H)(n, n')$. Third, let $n \in H'' \setminus H, n' \notin H''$ and $m \leq \mathsf{inf}(H'')(n)$. Since $n \in H'' \setminus H \Rightarrow n \in H_0$, Lemma 3.41 tells us that $m \triangleright \mathsf{cap}(H'')(n, n') = 0$. □

3.5 Conclusion

We have presented the flow framework, enabling local modular reasoning about recursively-defined properties over general graphs. Using the flow framework 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. The core reasoning technique has been designed to make minimal mathematical requirements, providing great flexibility in terms of potential instantiations and applications. We identified key classes of these instantiations for which we can provide existence and uniqueness guarantees for the fixpoint properties our technique addresses.

4 | Proof Technique and Automation

This chapter shows how to integrate flow interface reasoning into a standard separation logic. It presents a proof technique that is widely applicable, and illustrates its usage on a simple example program; the same core proof technique can be applied to all examples discussed in this chapter.

4.1 Proof Technique

Since flow graphs and flow interfaces form separation algebras, it is possible to define a separation logic (SL) using these notions as its *semantic model* (indeed, this is the proof approach taken by Krishna et al. [2018b]). By contrast, we encode flow interfaces within a standard separation logic without modifying its semantics. This has the important technical advantage that our proof technique can be naturally integrated with existing separation logics and verification tools supporting SL-style reasoning. In §4.4 we demonstrate this concretely for the Viper verifier, but our technique is also easy to extend to logics such as Iris which support (ghost) resources ranging over user-defined separation algebras [Jung et al. 2018b]. We focus here on the simple separation logic from §2.1 that does not have support for e.g. fine-grained concurrency, but our compatibility with standard SL semantics makes integration with more sophisticated logics straightforward.

4.1.1 Encoding Flow-based Proofs in SL

We now describe our proof technique, using the running example of the insertion procedure on a singly-linked list. While this is a simple example which can be handled by a wide variety of other techniques, we use it to illustrate the key points of our technique as it minimizes any example-specific complexity. As we show subsequently, by layering on more flow domains the same proof sketch extends to insertion into doubly-linked lists or the Harris list, illustrating the power of flow-based proofs to scale to algorithms with arbitrary traversals and complex overlaid structures.

Abstracting the Heap using Flow Interfaces The key idea behind encoding flow interfaces in SL is to use a special ghost field `intf` to store an interface for every node. The interface stored in this field at address x is the *singleton interface* of a flow graph containing only node x. While the flow framework gives us the power to use interfaces of any size in our proofs to reason

```
1  // Let edges({next: y}, x, z) := (z = y ≠ null ? λ_id : λ_0)
2  // Let γ_ls(x, {next: y}, m) := m = 1
3  // Let φ(I) := I^in = {h ↣ 1, _ ↣ 0} ∧ I^out = {_ ↣ 0}
4
5  method traverse(h: Node, X: Set[Node])
6    returns (X': Set[Node])
7    requires Gr(X, J, γ_ls) * φ(J_X)
8    ensures Gr(X', J', γ_ls) * φ(J'_X')
9  {
10   var l := h; var r := l.next;
11   while (nondet() && r != null)
12     invariant Gr(X, J, γ_ls) * φ(J_X)
13     invariant l ∈ X * r = next(f_l) * (r ≠ null ⇒ r ∈ X)
14   {
15     l := r; r := l.next;
16   }
17   {Gr(X_1, J, γ_ls) * Gr(X \ X_1, J, γ_ls) * V̄(J_X) * φ(J_X) * X_1 = {l}}
18   var X_1 := {l}; var X'_1 := insert(l: Node, X_1);
19   {Gr(X'_1, J', γ_ls) * Gr(X \ X_1, J, γ_ls) * J_X1 ≲ J'_X'1 * V̄(J_X) * φ(J_X) * X_1 = {l}}
20   return X ∪ (X'_1 \ X_1);
21 }
22
23 method insert(l: Node, X_1: Set[Node])
24   returns (X'_1: Set[Node])
25   requires Gr(X_1, J, γ_ls) * X_1 = {l}
26   ensures Gr(X'_1, J', γ_ls) * J_X1 ≲ J'_X'1
27 {
28   {Gr(X_1, J, γ_ls) * X_1 = {l}}
29   var r := l.next;
30   var n := alloc(); initNode(n, 0);
31   X'_1 := X ∪ {n};
32   {Gr(X_1, J, γ_ls) * N(n, f_n, J, J) * X'_1 = X_1 ∪ {n} * V̄(J_X'1) * J_X1 ≲ J_X'1}
33   n.next := r; l.next := n;
34   {✻_{x∈X'_1} (N(x, f_x, J, J') * γ_ls(x, f_x, J'_x^in)) * J_X'1 = J'_X'1 * V̄(J_X'1) * J_X1 ≲ J_X'1}
35   sync({l, n});
36   {✻_{x∈X'_1} (N(x, f_x, J', J') * γ_ls(x, f_x, J'_x^in)) * V̄(J'_X'1) * J_X1 ≲ J'_X'1}
37 }
```

Figure 4.1: A proof sketch using our flow-based proof technique for the insert procedure of a linked list.

about concrete states, this choice is in some sense canonical. If we tie the singleton interfaces to abstract the singleton heap regions, then we can express the interface of any set of nodes as the composition of the respective singleton interfaces. On the other hand, if we only tied the interface of a larger region to the heap, we would lose the ability to precisely reason about modifications to single nodes when needed (see Lemma 3.11).

ENCODING The basic building block of flow-based specifications is a node predicate $\mathsf{N}(x, \vec{\mathsf{f}}_x, J, J')$ that abstracts a node x to the corresponding singleton flow graph:

$$\mathsf{N}(x, \vec{\mathsf{f}}_x, J, J') \coloneqq x \mapsto \left\{\texttt{intf}\colon J_x, \vec{\mathsf{f}}_x\right\} * \mathsf{dom}(J_x) = \{x\} * \forall y.\ {J'_x}^{out}(y) = {J'_x}^{in}(x) \triangleright \mathsf{edges}(\vec{\mathsf{f}}_x, x, y)$$

Here, J and J' are logical variables of type $\mathfrak{N} \to \mathsf{FI}$, and we use the notation J_x for the interface that J maps node x to[1]. This predicate expresses the fact that we have a heap cell at address x, whose `intf` field stores the singleton interface J_x and whose other fields are captured by the parameter $\vec{\mathsf{f}}_x$ (a list of field-name/value mappings). N is parametric with a user-defined abstraction function edges that abstracts the field values (as defined by $\vec{\mathsf{f}}_x$) of x to the flow graph edge function for the pair of nodes (x, y); we constrain the outflow of x's singleton interface J'_x in terms of this edge.

The purpose of using two interface maps J_x and J'_x is to allow the fields of x to be abstracted by an interface that may temporarily differ from the one stored in `intf`. We call such nodes *dirty nodes*. This flexibility will be used in our proofs to model intermediate states between modifying the heap and updating the flow graph abstraction.

SPECIFICATIONS Specifications in our technique, for instance method pre and postconditions, are naturally expressed using the node predicate N within an iterated separating conjunction over a *set* of nodes X, to express a region of the heap abstracted by a flow interface:

$$\mathsf{Gr}(X, J, \gamma) \coloneqq \mathop{\Large *}_{x \in X} \left(\mathsf{N}(x, \vec{\mathsf{f}}_x, J, J) * \gamma(x, \vec{\mathsf{f}}_x, {J_x}^{in}(x), \dots)\right) * \overline{\mathcal{V}}\,(J_X)$$

$\mathsf{Gr}(X, J, \gamma)$ describes a set of nodes X, each of which satisfies N and none of whom are dirty (as we use $\mathsf{N}(x, \vec{\mathsf{f}}_x, J, J')$ with $J = J'$)[2]. J_X is syntactic sugar for the (iterated) interface composition $\bigoplus_{x \in X} J_x$; the assertion $\overline{\mathcal{V}}\,(J_X)$ represents the fact that the singleton interfaces stored in the `intf` fields of nodes in X compose to a valid interface.

The invariants of the data structure in question are encoded using a combination of the Gr predicate, and another predicate φ. Gr is parameterised by a user-specified predicate γ used to encode node-local properties, including constraints on the flow values of nodes. The predicate φ can instead be used to constrain the *composed* interface J_X, for instance expressing that it is a closed region with no outgoing edges.

[1] We assume here, for simplicity of presentation, that each address on the heap corresponds to a graph node, i.e., $\mathfrak{N} = \mathsf{Addr}$. This approach can easily be extended to the case where a graph node represents more than one heap address by replacing the $x \mapsto \{\texttt{intf}\colon J_x, \dots\}$ predicate with a user-defined predicate spatialRep that can be instantiated to specify all the addresses abstracted by graph node x.

[2] In specifications, we implicitly existentially quantify at the top level over free variables such as $\vec{\mathsf{f}}_x$, etc.

$$\mathsf{Gr}(X_1, J, \gamma) * \mathsf{Gr}(X_2, J, \gamma) * \overline{\mathcal{V}}\left(J_{X_1 \uplus X_2}\right) \equiv \mathsf{Gr}(X_1 \uplus X_2, J, \gamma) \qquad ((\text{De})\text{Comp})$$

$$\mathsf{N}(x, \vec{f}_x, J, J) * \gamma(x, \vec{f}_x, {J_x}^{in}(x), \dots) * \overline{\mathcal{V}}\left(J_{\{x\}}\right) \equiv \mathsf{Gr}(\{x\}, J, \gamma) \qquad (\text{Sing})$$

$$\mathit{true} \models \mathsf{Gr}(\emptyset, J, \gamma) \qquad (\text{GrEmp})$$

$$\overline{\mathcal{V}}\left(J_{X_1 \uplus X_2}\right) * X_1' \cap X_2 = \emptyset * J_{X_1} \precsim J'_{X_1'} * J_{X_2} = J'_{X_2} \models \overline{\mathcal{V}}\left(J'_{X_1' \uplus X_2}\right) * J_{X_1 \uplus X_2} \precsim J'_{X_1' \uplus X_2} \qquad (\text{Repl})$$

$$\overline{\mathcal{V}}\left(J_X\right) * x \in X \models {J_X}^{in}(x) + \_ = {J_x}^{in}(x) \qquad (\text{IntCompIn})$$

$$\overline{\mathcal{V}}\left(J_X\right) * x \in X * y \notin X \models {J_X}^{out}(y) = {J_x}^{out}(y) + \_ \qquad (\text{IntCompOut})$$

$$\overline{\mathcal{V}}\left(J_X\right) * Y \subseteq X \models \overline{\mathcal{V}}\left(J_Y\right) \qquad (\text{ValidSub})$$

Figure 4.2: Proof rules for proving entailments between flow-based specifications.

For example, consider the list insertion example shown in Figure 4.1; for this proof, we use the path-counting flow domain from Example 3.2. The abstraction function edges conditions on the `next` field of a source node and defines the corresponding graph edge to be the identity function λ_{id} if the field is non-null and the zero function λ_0 otherwise (we write $(P \ ? \ E_1 : E2)$ for C-style conditional expressions). The precondition of `traverse` on line 7 uses Gr with parameter γ_{ls}, to express that each node has a flow (i.e. path count) of 1, and φ to restrict the composite interface J_X to have inflow 1 at the head of the list h, inflow 0 everywhere else, and 0 outflow to all nodes. Thus, the precondition $\mathsf{Gr}(X, J, \gamma) * \varphi(J_X)$ implies that X contains a closed list rooted at h.

Proof Rules for Flows Our encoding of flow interfaces into SL and the predicates N and Gr give rise to generic proof rules for entailments (Figure 4.2). ((De)Comp), (Sing), and (GrEmp) all follow directly from the definition of Gr and basic properties of iterated separating conjunction. (Repl) is derived from the Replacement Theorem (Theorem 3.18), and we discuss it further below. (IntCompIn) and (IntCompOut) are properties of composite interfaces that follow directly from the definition of interface composition. Finally, (ValidSub) expresses the fact that if an interface J_X is valid (i.e. the implicit composition $\bigoplus_{x \in X} J_x$ is defined) then so is any sub-interface.

Note the connection between ((De)Comp) and the algebraic laws of standard inductive predicates such as lseg. For instance by combining ((De)Comp) and (Sing) we can prove the following rule to fold or unfold the graph predicate:

$$\mathsf{Gr}(X \uplus \{x\}, J, \gamma) \equiv \left(\mathsf{N}(x, \vec{f}_x, J, J) * \gamma(x, \vec{f}_x, {J_x}^{in}(x), \dots)\right) * \mathsf{Gr}(X, J, \gamma) * \overline{\mathcal{V}}\left(J_{X \uplus \{x\}}\right) \qquad ((\text{Un})\text{Fold})$$

However, an advantage of using the flow graph predicate Gr is that these are generic rules that apply regardless of the data structure they describe (the rules are parametric with γ) and these rules are proved once and for all here. By contrast, for every new inductive predicate and direction of traversal one would need to prove a new rule like ((Un)Fold).

The rule (Repl) is derived from the Replacement Theorem by letting $I = J_{X_1 \uplus X_2}, I_1 = J_{X_1}, I_2 = J_{X_2}$ and $I_1' = J_{X_1'}'$. We know $I_1 \precsim I_1'$ by $J_{X_1} \precsim J_{X_1'}'$, $\overline{\mathcal{V}}\left(J_{X_1 \uplus X_2}\right)$ tells us that $I = I_1 \oplus I_2$, and $X_1' \cap X_2 = \emptyset$ gives us $I_1' \cap I_2 = \emptyset$. The final condition of the Replacement Theorem is to prove is that there is no outflow from X_2 to any newly allocated node in X_1'. However, in a garbage collected environment this condition is true so long as we additionally restrict the abstraction function edges to only propagate flow along an edge (n, n') if n has a (non-ghost) field with a reference to n'. We assume a garbage collected environment in the proofs in this section, and in our Viper implementation in §4.4.

Method Calls and Frame Reasoning The list insertion procedure in Figure 4.1 is structured into two methods: the `traverse` method iterates through the list and stops at a nondeterministic point l (this is for simplicity; one could select a position based on value criteria, etc.). It then calls the `insert` method, which creates and inserts a new node n after l.

`traverse` sets a variable l to be the head and reads its `next` field into r before hitting a loop. The loop's invariant has the same spatial component as the precondition, but additionally says that l is in X, r is l's `next` field, and if $r \neq \mathit{null}$ then it is also in X (line 13). This is true initially, and since the definition of edges implies that l has an outflow of 1 to r, and the outflow of J_X is always 0, we can conclude that walking `next` fields must stay within X.

After the loop, the code calls the `insert` method. The precondition of `insert` (line 25) states that it operates on the singleton set of nodes $X_1 = \{l\}$; using ((De)Comp) we can show that the loop invariant implies this precondition. However, after `insert` returns, not only has l's interface potentially changed to a new interface J_l', but the set of nodes has potentially increased to X_1'. To establish the postcondition of `traverse`, we need to show that the interface of X_1' composes with the interface of the frame $X_2 := X \setminus X_1$ resulting in a valid interface satisfying φ.

To perform this frame reasoning, we use the rule (Repl). We know $\overline{\mathcal{V}}\left(J_{X_1 \uplus X_2}\right) = \overline{\mathcal{V}}\left(J_X\right)$ from the precondition of `traverse` and, from the postcondition of `insert`, that $J_{X_1} \precsim J_{X_1'}'$. By standard separation logic framing, we know that any new nodes in X_1' cannot overlap with any of the nodes in X_2. (Repl) allows us to deduce that $J_{X'} = J_{X_1'}' \oplus J_{X_2}$ is a valid interface and $J_X \precsim J_{X'}$. By setting $J' = J$ for all nodes in X_2, and checking that $\precsim$ preserves the conditions in φ, we obtain the postcondition of `traverse` in line 8.

A similar argument is used in the proof of `insert`. After reading l's `next` field into r, the code allocates a new node at address n using the inbuilt heap allocation procedure `alloc`. It then calls a helper method `initNode` that initializes n's `intf` field and has the following specification (its proof is omitted here, but can also be discharged with our technique):

$$\left\{n \mapsto \_\right\} \texttt{initNode}(n, m) \left\{\mathsf{N}(n, \vec{\mathsf{f}}_n, J, J) * I_\emptyset \precsim J_n * {J_n}^{in} = m\right\}$$

The postcondition gives us a graph node $\mathsf{N}(n, \vec{\mathsf{f}}_n, J, J)$ whose singleton interface J_n is a contextual extension of the empty interface $I_\emptyset$. Contextual extension forces n to have no outflow, but since n is a new node we determine n's inflow by the method argument m. In the case of `insert`, since we expect to satisfy the flow invariant γ_{ls} by linking n into the list, we do not need any external inflow to n, so we pass 0. Again, following our proof technique, we use (Repl) to lift

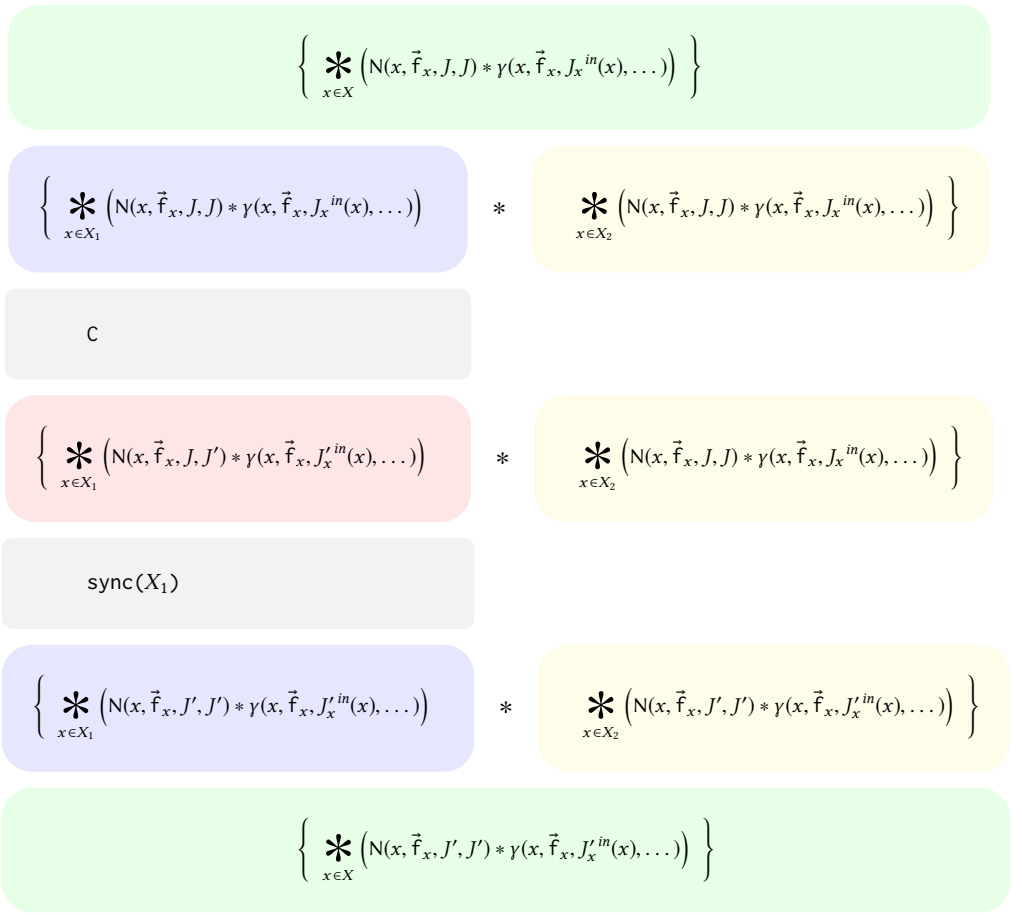


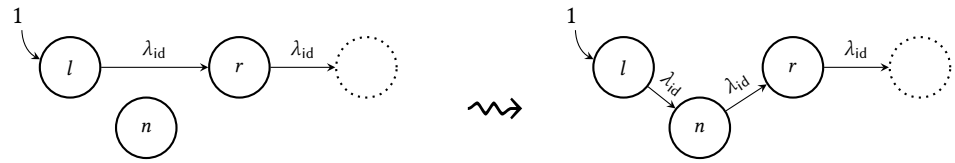
Figure 4.3: The crux of flow-based proofs: reasoning about a straight-line fragment of code C.

the contextual extension guaranteed by `alloc` to the current footprint, $J_{X_1} \precsim J_{X'_1}$, obtaining the intermediate assertion shown on line 32.

Reasoning about Modifications When the program we are reasoning about makes changes to the heap, the modified nodes become *dirty*: for some nodes we hold $\mathsf{N}(x, \vec{\mathsf{f}}_x, J, J')$ with $J_x \neq J'_x$. Eventually, we need to update the `intf` fields of all such dirty nodes, to once more obtain a state in which the heap is in sync with the flow abstraction. We add a special procedure sync to update the flow abstraction of a given *dirty region* D:

$$\left\{ \circledast_{x \in D} \mathsf{N}(x, \vec{\mathsf{f}}_x, J, J') * J_D = J'_D \right\} \ \mathsf{sync}(D) \ \left\{ \circledast_{x \in D} \mathsf{N}(x, \vec{\mathsf{f}}_x, J', J') \right\}$$

The precondition of sync checks that the interface of D is preserved, using the fact that the `intf` fields of these nodes store the *previous* interfaces which composed to form an interface for D. By Lemma 3.16, we know that for this condition to be satisfied, the dirty region that one syncs must be a superset of the flow footprint of the modification.



$J_{\{l,n\}} = (\{l \rightarrowtail 1, n \rightarrowtail 0\}, \{r \rightarrowtail 1\})$ $\quad$ $J'_{\{l,n\}} = (\{l \rightarrowtail 1, n \rightarrowtail 0\}, \{r \rightarrowtail 1\})$

Figure 4.4: Inserting a new node n into a list between existing nodes l and r. Edges are labeled with edge labels for path counting (λ_0 edges omitted), and curved arrows indicate inflow. The interfaces of the dirty region $\{l, n\}$ before and after the modification are shown below.

This is the crux of our proofs: reasoning about the modifications made by a straight-line code fragment `C` with no method calls. Figure 4.3 shows our proof technique for showing that a generic flow-based specification (the green first and last lines of Figure 4.3) is preserved by the program `C`. The specification is essentially $\mathrm{Gr}(X, J, \gamma)$, but we expand it in the figure for clarity. The first step is to decompose the region X (using the rule ((De)Comp)) into X_1, the blue region modified by `C`, and X_2, the yellow unmodified frame. The program `C` then modifies X_1, and this is reflected in the intermediate specification in the line after `C` as we go from $\mathsf{N}(x, \vec{\mathtt{f}}_x, J, J)$ to $\mathsf{N}(x, \vec{\mathtt{f}}_x, J, J')$. This is the dirty state; accordingly, this region is now colored red. Before we can call sync, we need to show that the nodes in X_1, and their new interfaces J', satisfy the program-specific invariant γ, and (by the precondition of sync) that the interface of X_1 is the same ($J_{X_1} = J'_{X_1}$). If these conditions are satisfied, sync will write the new interfaces given by J' to the `intf` field of nodes in X_1, updating the state to the intermediate specification in the line after `sync`(X_1) where the X_1 region is once more in sync (blue). Finally, we can once again use ((De)Comp) to compose X_1 and X_2 and obtain the postcondition.

To see this proof technique in action, consider the proof of `insert`. After allocating a new node n, the code manipulates the pointers to link n into the list right after l. We now have a state in which both l and n are dirty: due to the heap modification, their fields abstract to different interfaces from those still stored in their `intf` fields. The dirty region before and after modification is shown in Figure 4.4. We must now show that there exists a flow in the dirty region $\{l, n\}$, by exhibiting flow values for these nodes that satisfy the flow equation for the new edge functions in this region. In this case, we can see that there exist flow values $\{l \rightarrowtail 1, n \rightarrowtail 1\}$ that satisfy the flow equation in the dirty region. Moreover, we can show that the flow invariant γ_{ls} is true given these flow values, and we can calculate the outflow of this region and show that it matches that of the desired interface. We thus obtain the intermediate state on line 34, where the new interfaces J'_x have their outflow components computed from the heap by edges, and inflow components equal to the flow values that we computed (recall that inflows for singleton interfaces are equal to the nodes' flow values). We can now call sync, which checks that the dirty region's interface is indeed preserved and writes the new interfaces into the `intf` fields of all nodes in the dirty region, resulting in the state described on line 36. Following our proof technique for handling procedure calls, we follow this with an application of the Replacement Theorem, which in this case is a vacuous call since the frame X_2 is empty. We thus obtain the postcondition, completing our proof.

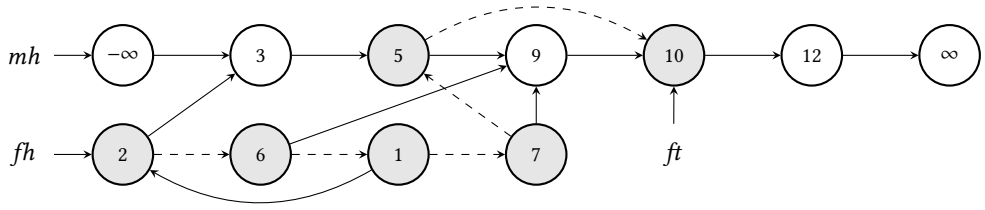


Figure 4.5: A potential state of the Harris list with explicit memory management. `fnext` pointers are shown with dashed edges, marked nodes are shaded gray, and null pointers are omitted for clarity.

Effective Acyclicity The proof above used the path-counting flow domain, but did not restrict itself to effectively acyclic (EA) flow graphs. Thus, the precondition in line 25 does not imply that X contains *only* a list rooted at h; the flow equation in general has multiple fixpoints, and in particular allows X to contain cycles of nodes with flow 1 unreachable from h. To rule out such undesirable cases, we can (as discussed in §3.4.3) use the EA restriction of the path-counting domain.

In this case, one needs to perform two extra checks when syncing the dirty region. First, to prove that a flow exists for the modified dirty region, we check that the dirty region does not contain any cycles (simple, in this case). Second, we must check that the change to the dirty region is a subflow-preserving extension (c.f. Definition 3.37) to rule out the possibility of creating cycles in the larger graph. We omit the full details here, though §4.4.1.1 shows how we automate such checks for our case studies.

4.2 Extending To The Harris List

The power of flow-based reasoning is exhibited when we try to extend this proof to one that inserts a node into an overlaid data structure such as the Harris' list, a concurrent non-blocking linked list algorithm [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. As with the sequential linked list, 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.

In a non-garbage-collected environment, unlinked nodes cannot be immediately freed as there may be suspended threads continuing to hold a reference to them. A common solution is to maintain a second "free list" to which marked nodes are added before they are unlinked from the

main list (this is the so-called drain technique). These nodes are then labeled with a timestamp, which is used by a maintenance thread to free them when it is safe to do so. This leads to the kind of data structure shown in Figure 4.5, 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 non-trivial 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 `next` edges from nodes in the free list point to nodes in the free list or main list.

(c) All nodes in the free list are marked.

(d) ft is an element in the free list.

Fortunately, with flows, we can easily adapt the proof of the singly-linked list to one that proves that the above invariants are maintained by Harris' algorithm. To do so, we use the product of two path-counting flow domains: one to track the path count from the head of the main list, and one from the head of the free list. The definitions of edges, γ, and φ require analogous changes:

$$\begin{aligned}\text{edges}(\{\texttt{next}\colon y, \texttt{fnext}\colon z\}, x, v) \coloneqq (v = \mathit{null}\ ?\ \lambda_0 \\ : (v = y \wedge y \neq z\ ?\ \lambda_{(1,0)} \\ : (v \neq y \wedge y = z\ ?\ \lambda_{(0,1)} \\ : (v = y \wedge y = z\ ?\ \lambda_{\mathsf{id}} : \lambda_0))))\end{aligned}$$

$$\begin{aligned}\gamma_{\mathsf{h}}(x, \{\texttt{next}\colon y, \texttt{fnext}\colon z\}, m) \coloneqq (m = (1,0) \vee m = (0,1) \vee m = (1,1)) \\ \wedge (m \neq (1,0) \Rightarrow M(y)) \\ \wedge (x = ft \Rightarrow m \neq (1,0))\end{aligned}$$

$$\begin{aligned}\varphi(I) \coloneqq I^{in} = \{mh \rightarrowtail (1,0), fh \rightarrowtail (0,1), \_ \rightarrowtail (0,0)\} \\ \wedge\ I^{out} = \{\_ \rightarrowtail (0,0)\}\end{aligned}$$

Here, edges encodes the edge functions needed to compute the product of two path counting flows, the first component tracks path-counts from mh on `next` edges and the second tracks path-counts from fh on `fnext` edges ($\lambda_{(1,0)} \coloneqq (\lambda(m_1, m_2).\ (m_1, 0))$ and $\lambda_{(0,1)} \coloneqq (\lambda(m_1, m_2).\ (0, m_2))$). The node-local invariant γ_{h} says: the flow is one of $\{(1,0), (0,1), (1,1)\}$ (meaning that the node is on one of the two lists, invariant (a)); if the flow is not $(1,0)$ (the node is not only on the main list, i.e. it is on the free list) then the node is marked (indicated by $M(y)$, invariant (c)); and if the

node is *ft* then it must be on the free list (invariant (d)). The constraint on the global interface, φ, says that the inflow picks out *mh* and *fh* as the roots of the lists, and there is no outgoing flow (thus, all non-null edges from any node in the graph must stay within the graph, invariant (b)).

Moreover, the proof outline remains essentially identical. All the complexity of reasoning about the various cases of possible overlap are reduced to our standard requirement of showing that the flow interface of the modified region is preserved. This is tedious to perform and present in a by-hand proof, so we omit it here for clarity of presentation. Nevertheless, as we show in §4.4 and in our evaluation (§4.6), this portion of the proof is can be fully automated, putting no additional burden on the end-user. We note that our proof of Harris' list is performed in a sequential separation logic. However, the invariant we verify is sufficient to verify the same algorithm in a concurrent setting, and corresponds to the one used by Krishna et al. [2018b].

4.3 The Edge-local Flow Transformation

In the example proof in Figure 4.1, it was easy to show that the modification that linked the new node n into the list preserved the interface of some region X_1' because the flow footprint of the modification was small (in this case, equal to the set of modified nodes $\{l, n\}$). However, in general, a particular flow domain and heap modification may yield a larger (and potentially unbounded) flow footprint. Proving that an unbounded dirty region has a solution to the flow equation, and preserves a prior interface becomes challenging, and generally requires precise information about the graph structure, along with ad-hoc inductive reasoning. This section presents a technique to *transform* any flow domain into one in which the flow footprint has a well-defined structure and moreover always admits a solution to the flow equation. Our technique breaks down the task of reasoning about an unbounded flow footprint into a series of reasoning steps about smaller, more structured regions, facilitating easier and more natural proofs of programs such as the PIP example.

To motivate the problem, consider indeed the flow domain for the PIP introduced in §3.1; we employ multisets of (strictly positive) integers to capture the priorities of each node and its predecessors. Recall the situation in which a process p attempts to acquire a resource r, adding an edge (p, r) to the PIP graph. In general such a modification has an unbounded flow footprint, in the worst case comprising all nodes reachable from r (e.g., consider p_1 and r_1 in Fig. 3.1).

Constructing proofs in the presence of such unbounded flow footprints (dirty regions, in the proofs) makes it difficult to show, after a modification, that the flow footprint can be abstracted to a valid flow graph matching its prior interface: the key requirement of our local reasoning technique. Demonstrating that the flow equation has a solution in an unbounded dirty region in general requires intimate knowledge of the graph structure and field values within this region. Since the entire region must be synced at once, this knowledge must typically cross at least one specification boundary (e.g. a loop head), adding complexity and length to user specifications and impeding proof automation. Most seriously, for *proving* this fact, one would have to revert to complex inductive reasoning about fixpoints over general graphs – the very problem that flow-based reasoning is intended to avoid.

Our solution to this problem is to transform the flow computation into an one that can be

computed simply and incrementally. We do this by transforming the *flow domain* to one which is edge-local; and modifying any given flow graph by replacing its edge functions with corresponding *constant* functions providing the same flow values. More precisely, for any flow domain $(M, 0, +, E)$, we define $E_M \coloneqq \{(\lambda\_.\ m) \mid m \in M\}$ to be the set of constant edge functions. The flow domain $(M, 0, +, E_M)$ is an edge-local flow domain (see §3.4.1), which means any graph over this domain always has a unique flow. The following lemma shows that any flow graph H over the original flow domain can be transformed into a flow graph H' over the corresponding edge local domain and vice versa.

Lemma 4.1. *Let $(M, 0, +, E)$ be a flow domain, $G = (N, e)$ be a graph over this domain, and $G' = (N, e')$ a graph over the corresponding edge-local flow domain $(M, 0, +, E_M)$. Then given in, $\mathsf{flow} : N \to M$, the following statements are equivalent*

(i) $H = (N, e, \mathsf{flow})$ *is a flow graph with inflow in*

(ii) $H' = (N, e', \mathsf{flow})$ *is a flow graph with inflow in and* $\forall n, n'.\ e'(n, n') \coloneqq (\lambda\_.\ \mathsf{flow}(n) \triangleright e(n, n'))$

We can lift this edge-local transformation to our proof technique as follows. We define a new field `flw`, and redefine the original edge functions to propagate (from each source node) the value in its `flw` instead its the actual flow value, using the following abstraction function:

$$\mathsf{edges}'(\left\{\texttt{flw}\colon f, \vec{\mathsf{f}}_x\right\}, x, y) \coloneqq (\lambda\_.\ \mathsf{edges}(x, \vec{\mathsf{f}}_x)(y)(f))$$

where edges is the original abstraction function. Now, if all nodes in a flow graph satisfy an *additional invariant* that their flow values match their `flw` fields, then by Lemma 4.1 the flow values in this transformed program are also a solution to the original fixpoint equation. Effectively, we *decouple* the transitive propagation of flow values, via the additional field `flw`.

The key advantage of this transformation is that the only node whose flow value can change when an edge (n, n') is modified is the single target node n'. Any outgoing edge from n' propagates the same value that it used to, because the `flw` field of n' has not changed; in other words, the flow footprint for any single edge modification is at most the two nodes that the edge connects. Of course, the invariant that the `flw` of every node matches its flow may be broken by such an update, but only (immediately) for node n'. Crucially, this discrepancy does not prevent us from calling `sync` in our proofs; the node whose `flw` value is out of sync with its actual flow values can now be tracked in pure specifications.

This edge-localisation technique lends itself to natural proofs of programs whose *algorithms* reflect this break-then-notify style. For instance, returning to the PIP; here is what the key proof definitions look like after our edge-local transformation:

$$\vec{\mathsf{f}} \coloneqq \{\texttt{flw}\colon S, \texttt{curr\_prio}\colon q, \texttt{next}\colon y, \dots\} \qquad \mathsf{edges}(\vec{\mathsf{f}}, x, z) \coloneqq (z = y \neq \mathit{null}\ ?\ (\lambda\_.\ \{q\}) : \lambda_0)$$

$$\gamma_{\mathsf{pip}}(x, \vec{\mathsf{f}}, m) \coloneqq S = m \wedge q = \max(S)$$

If a state of the PIP satisfies the assertion $\mathrm{Gr}(X, J, \gamma_{\mathsf{pip}})$ then γ_{pip} enforces that the value stored in the `flw` is equal to the flow, and edges specifies that the outflow to the `next` node is the maximum of the priorities given by the flow (this is also stored in the `curr_prio` field).

What is interesting is that the code in Figure 3.1 already stores the flow in the `prios` field, and the priority going from one node to the next is this locally cached flow value, meaning that our edge-local flow mirrors the algorithm. Moreover, after modifying an edge, the algorithm calls `update` to correct the target node, which only updates the `prios` field of that one node. There is a recursive call to `update` to fix any downstream nodes whose flow may have changed, but the state before this recursive call can be described more naturally using the edge-local flow as one where every node except `n.next` satisfies γ_{pip}, and for `n.next` the required update is to remove the priority `old_prio` and add the priority `n.curr_prio` from its `prios` field (which is also the flow given to `n`).

The edge-local transformation is applicable to any flow domain and program; though it is most useful in cases when the program makes modifications with unbounded flow footprints and then proceeds to iteratively fix the invariant in this region. A potential downside of using the edge-local transformation is that it does not play well with effective acyclicity (EA), for, by definition, no non-trivial edge-local flow graph is EA. For instance, if one were using the path-counting flow domain coupled with the EA restriction to describe a tree, then the edge-local transformation of this domain would allow extra nodes that formed unreachable cycles (see Figure 3.2(b)).

The transformation itself is just a matter of moving one part of the flow computation from the meta-theory to the node-local invariants. The flow of the transformed graph only corresponds to the original flow when all nodes satisfy an additional invariant. Despite the fact that the result of this transformation appears to have gotten rid of any global reasoning, this does not make the flow framework redundant. The framework still provides a means of dealing with unbounded aggregation of incoming flow values. For example, if one were to reason about the edge-local version of the PIP flow without the flow framework, one would need to axiomatize the maximum operator over values from all nodes in the graph and use ad-hoc rules to deduce that certain modifications only affect the priority of neighbouring nodes. On the other hand, the flow framework provides a uniform means to reason, in a manner compatible with SL-style framing, about everything from complex global properties to simple local reasoning about edge-level aggregation.

4.4 Proof Automation

In this section, we develop upon our proof strategy to enable automated checking of flow-based proofs. Our checking simulates a translation from source-level programs annotated with flow specifications, into Viper [Müller et al. 2016] programs whose successful verification implies the existence of a source-level proof. Our goal is for the *only* requirements on the user to be the definition of the flow domain instantiation (with a chosen labelling function lifting heaps to graphs), and the classical specifications for deductive reasoning (pre/post conditions and loop invariants); essentially only the blue annotations in Figure 4.1. We show how to automate the two main classes of flow domain introduced in §3.2: edge-local flow domains, and effectively-acyclic (EA) graphs.

The flow-based proof technique presented in §4.1 affords a great deal of flexibility in the construction of correctness proofs, which was key to its general applicability. Unfortunately, this

flexibility requires creativity on the part of the proof author, and presents a number of challenges for achieving the high degree of automation that we aim for:

(C1) When do we call sync on modified regions? Our proof technique allows us to sync the flow interfaces with the heap at any point and any number of times. To have an automated proof technique, we need to find a strategy that works for a large class of examples.

(C2) Which dirty regions do we sync? As we saw in §4.1, we must sync a superset of the flow footprint of any modification. As the flow footprint is defined in terms of the flow of the larger graph, it is not always possible to compute the flow footprint in a local and automatable manner. We need a heuristic for choosing a suitable dirty region for a given call to sync.

(C3) When we call sync on a given region, how do we prove that it has a flow and that its interface is preserved? We have seen that certain classes of flow domains (§3.4) guarantee the existence of the flow. However, to express the flow equation (FlowEqn) in the dirty region and to derive its old and new flow interfaces requires sum comprehensions. Furthermore, for the effectively acyclic class, one additionally requires a computation over all paths within the region to establish subflow preservation (Definition 3.37).

(C4) Finally, how do we automatically infer relations between the interfaces of different sizes? In the example proof in §4.1.1, one had to call the Replacement Theorem after the method call to `insert` in order to relate the interface $J'_{X'_1}$ of `insert`'s footprint to the interface $J'_{X'}$ of the caller. One also needs to relate singleton interfaces to the interfaces of larger, potentially unbounded, regions containing them. This reasoning depends on applying the definition of interface composition to an unbounded region, something that is hard to do automatically. We need a technique to apply such proof steps to the relevant interfaces at appropriate points.

We present solutions to all of these challenges in the remainder of this section, under certain restrictions which we explain next.

Restrictions Given that automated reasoning about inductive properties such as reachability over unbounded graphs, that can be encoded as flows, are known to be undecidable [Immerman et al. 2004], we must sacrifice completeness for a sound automated proof technique. Our automation story (unlike the by-hand proof technique of the previous section) thus works under some additional restrictions: (1) we require that the proof uses either an edge-local flow domain or the EA restriction; (2) we require that pre/post-condition specifications are each written in terms of node-local conditions γ on each node's concrete fields and flow, and a constraint φ on the global interface of the subgraph on which a method operates (i.e., we don't automate support for other combinations of interfaces in specifications); and (3) our calculation of dirty regions is heuristic, and relies on one of our techniques being employed to localise the flow footprint of each modification to a finitely-bounded (not necessarily statically-known) sub-region of the graph.

As we saw in §4.3, any flow domain can be transformed into an edge-local one that has the same set of solutions to the flow equation. Thus any proof can be done with edge-local flows at the cost of more reasoning steps or ghost code; in many cases (e.g. the PIP, Dijkstra, and Composite examples) however, these steps match those of the code itself. While in theory one could write specifications about multiple potentially-overlapping interfaces of the data structure, we have not yet found an algorithm where this is necessary. Finally, our dirty region calculation heuristic does rule out certain modifications that have an unbounded flow footprint. However, our edge-local transformation can again help tame the flow footprint and bring it within reach of our heuristic. Note that for fine-grained concurrent algorithms, our requirement to keep the flow footprints local corresponds to locality of atomic updates; any node whose flow invariant is broken for longer would have to be locked to prevent other threads observing inconsistent states. As we demonstrate in our evaluation (§4.6), we are able to capture a wide variety of data structures, flow domains and properties within these restrictions.

When to Sync Our aim of requiring minimal and simple annotations from the user guides our approach to Challenge (C1). To avoid requiring user-specifications about dirty regions (which, when later synced, will require fine-grained knowledge of the graph structure and flow values), we perform a sync before each *verification boundary*, meaning before a loop or method call, and at the end of a loop or method body. This design naturally organises the verification of a method into *phases*; we begin with a consistent graph and interfaces (and an empty dirty region), make modifications which add some nodes to the dirty region, and then sync these before the next verification boundary back to a consistent interface.

Which Region to Sync Since we only sync at verification boundaries, we only need to build up a suitable dirty region between each such boundary. Our heuristic for maintaining this dirty region is as follows: every time the field of an object is modified in the program code, we add that object to the dirty region. As discussed above, this might not work for some input programs and flow domains, but constructions such as the edge-local transformation of §4.3 can be employed to overcome this limitation. Combined with our restrictions, this gives us a way to solve Challenge (C2) for a wide variety of challenging examples. We describe how we deal with Challenges (C3) and (C4) in the relevant parts of the description of our automation procedure in the next subsection.

4.4.1 Automatic Generation of a Flow-based Proof

We now describe our automation technique as a syntactic translation from minimally-annotated programs to Viper files that perform a proof in our proof technique from §4.1.

Viper's Logic The core logic employed by the Viper verification infrastructure is based on the *implicit dynamic frames* logic [Smans et al. 2009], a close relative of separation logic; the embedding of core separation logic is known [Parkinson and Summers 2012]. Viper natively supports expressing and automatically reasoning about iterated separating conjunctions [Müller et al.

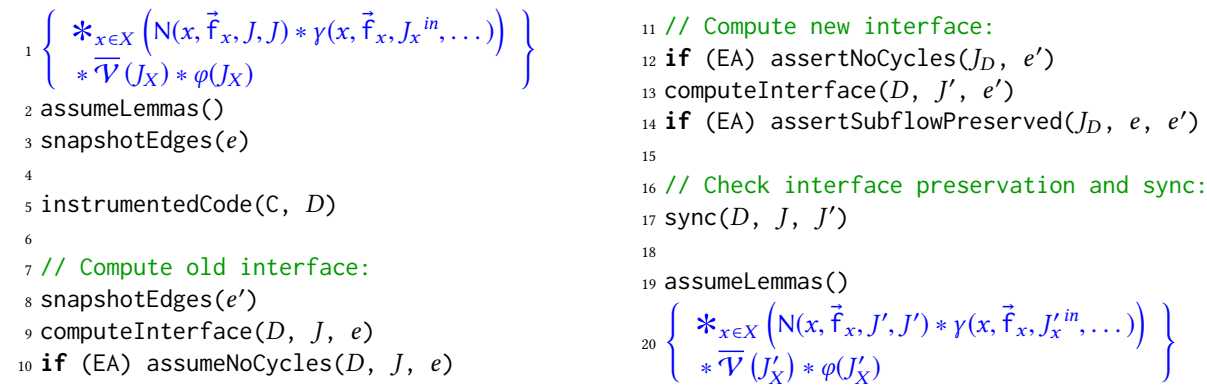


Figure 4.6: The structure of annotated Viper programs generated by our technique.

2016]. This is why we chose Viper for demonstrating our automation techniques, but any tool with similarly automated support for iterated separating conjunctions and custom mathematical types/functions could in principle be used in its place. Our encodings could also be adapted to less automated settings; for a more manual proof, our heuristically-chosen dirty regions could be overridden by manual specification; our Viper encoding also supports user-selected additions to this dirty region.

User Input Our automation relies on the following ingredients as inputs from the user:

- A definition of the flow domain in question, along with the choice of whether this employs edge-local flows, or effectively-acyclic graphs.
- A labelling function edges(x, y), defining, for each pair of references x and y, the edge function between their corresponding graph nodes, in terms of the concrete field values of x.
- Pre/post-conditions and loop invariants in terms of both node-local conditions (γ from §4.1) and constraints on the interface of the composite graph in question (φ from §4.1).

Generation Procedure Figure 4.6 shows the structure of the annotated output Viper program generated by our technique. The input to this generation is a verification phase, i.e. a piece of straight-line code `C` not containing any loops or method calls, and its pre/post-condition specifications (shown in lines 1 and 20 respectively). The output is an annotated Viper program fragment that tries to prove that `C` satisfies the flow-based specification of this phase, using the proof technique shown in Figure 4.3. Note that as allocation is done by a method call to `alloc`, `C` cannot modify the footprint, so the same set of nodes X is used in both specifications (handling method calls is described in a subsequent paragraph).

We now show how we instrument `C` to compute the dirty region as per our heuristic, compute the new flow graph for this region, and check that it preserves its interface so that we can sync

the region and prove the postcondition of this phase. We achieve this using the following series of auxiliary steps:

1. The first and last components (lines 2 and 19) are to assume certain flow interface lemmas that relate interfaces of different regions; these will be described in detail in §4.4.1.2.

2. We take a snapshot of the graph edges before and after the instrumented code (line 5). That is, we define auxiliary variables e and e' respectively to be equal to the edge functions of the graph that abstracts the heap (as given by the edges function) at the code locations in question.

3. We instrument the code `C` to compute the set of dirty nodes D prescribed by our heuristic. We do this by following every field write command with a command that adds the source node to the set D, the resulting code is denoted `instrumentedCode(C, D)`.

4. We know from the precondition that all the singleton interfaces J_x for $x \in D$ composed to a valid interface. Thus, we can compute the old interface of D, J_D, in terms of the singleton interfaces J_x, denoted `computeInterface(D, J, e)` (described in §4.4.1.1). Moreover, if we are in the EA case, we can also assume that the old flow graph was EA (line 10), which additionally constrains the singleton interfaces J_x and edges e.

5. Before we can compute the new interface of D, we must first check that D has a flow. In the edge-local case, this is always true, but in the EA case we additionally check that the new edges e' have not introduced any cycles inside D (`assertNoCycles(`J_D`, e')`). We can now use `computeInterface(D, J', e')` to compute the new interface J'_D of D and relate it to the new singleton interfaces J'_x for $x \in D$. Additionally, to ensure in the EA case that we do not introduce new cycles in the larger region X, we also check that D's interface is subflow preserving (Definition 3.37).

6. Finally, we try to sync D. This step checks that the old interface of D, $J_D = J'_D$, the new interface of D.

This generation relies on several macros (`computeInterface`, `assertNoCycles`, etc.) that are not easy to encode as Viper assertions because they contain arbitrary sums or reasoning about all paths through a region. We next describe how we encode such computations in an automatable manner.

4.4.1.1 Computing Interfaces and Flows

Solving challenge (C3), computing the flow and flow interface of a region, boils down to solving two problems: automating the sum of a quantity over a statically-unknown set of nodes D, and automating sums over all paths through a region D.

Given a region D, where each $x \in D$ has interface J_x, and the interface of D is $J_D \coloneqq \bigoplus_{x \in D} J_x$, we know that the outflow of D is given by ${I_D}^{out}(y) = \sum_{x \in D} {J_x}^{out}(x, y)$. Computing this sum automatically is hard because in many programs, the size and contents of D are statically unknown

(certain nodes could only be modified in some branches of the code, may or may not alias one another, etc.). Directly instantiating the flow equation (FlowEqn) is also hard for the same reason. We solve this issue by amending `instrumentedCode(C, D)` to also track the set of nodes D as a *mathematical list* D_L containing each node in D exactly once (the list's contents may depend on branch conditions, aliasing, etc., in a way known only to the prover). We then axiomatize a function to simulate a functional program to compute the sum over these terms, using E-matching [Detlefs et al. 2005] in place of pattern-matching, to automate this computation via quantifier instantiation *within the SMT solver* used by Viper to verify the instrumented program.

In the EA case, checking effective acyclicity (Definition 3.33) of the modified graph and showing that the modified interface is subflow-preserving (Definition 3.37) requires computing the capacity of the (old and new) dirty region, which is a sum over all paths through D. To enumerate these paths, we define the following helper function that approximates the capacity of the dirty region, where m, n are nodes and D_L is a set of nodes:

$$\begin{aligned} \mathsf{capAux}_e(m, n, \emptyset) &= e(m, n) \\ \mathsf{capAux}_e(m, n, n' :: D_L) &= \mathsf{capAux}_e(m, n, D_L) + \mathsf{capAux}_e(m, n', D_L) \circ \mathsf{capAux}_e(n', n, D_L) \end{aligned}$$

$\mathsf{capAux}_e(m, n, D_L)$ is inspired by the Floyd-Warshall algorithm, and contains terms for all simple paths from m to n using nodes in D_L (and some other cyclic paths, but those will be zero by effective acyclicity). When a flow graph $H = (N, e, \mathsf{flow})$ is EA (for instance, we know this in the pre-state), we can assume that for every $n \in N$, $\mathsf{flow}(n) \triangleright \mathsf{capAux}_e(n, n, D_L \setminus \{n\})$. And when we check that the new dirty region is effectively acyclic, since we do not yet know the new flow values, we use the following lemma:

Lemma 4.2. *Let (M, E) be a flow domain such that M is a positive monoid and E is a reduced set of endomorphisms, $G = (N, e)$ be a graph over this domain, and $in\colon N \to M$ be an inflow for this graph. If for every $n, n' \in N$, $in(n) \triangleright \mathsf{capAux}_e(n, n', D_L \setminus \{n, n'\}) \triangleright \mathsf{capAux}_e(n', n', D_L \setminus \{n'\}) = 0$, then there exists $\mathsf{flow}\colon N \to M$ such that $(N, e, \mathsf{flow}) \in \mathsf{FG}$ is a flow graph and is effectively acyclic.*

4.4.1.2 Flow Interface Lemmas

The problem to solve in Challenge (C4) is how to axiomatize properties of flow interfaces and their compositions over unbounded regions, and how and when to apply them. It is again not possible to have a complete algorithm to decide properties of flow interfaces, so our approach is to instead select a number of core lemmas about flow interfaces that are typically required, and axiomatise them in Viper. The lemmas concern the interfaces stored in the `intf` fields of nodes; some rely on these corresponding to the current flow values of the nodes. We apply these lemmas only in consistent states, (indicated by `assumeLemmas()` in Figure 4.6) at the beginning and end of each verification phase, so that they apply both to already-established and newly-established interfaces arising in our proofs. These lemmas include the Replacement Theorem, which is used to reason about interface composition after a method call (see §4.1.1), which we apply at the beginning of the subsequent verification phase.

4.4.1.3 Soundness

The soundness of our automated flow-based proof technique relies on the soundness of the meta-theory of the flow framework. This consists of the basic theory of flows (§3.2.1) and the flow-based proof rules from Figure 4.2, that have been described and proved in this dissertation (see also Appendix A.1 for the flow framework encoding and lemmas we use in Viper). We note that these proofs have not been mechanized yet, and we are currently working on mechanizing the flow framework in Coq. Apart from the flow framework, the rest of our technique is simply a syntactic transformation from a restricted class of input programs to annotated Viper programs with an analogous underlying semantic model; thus soundness of the separation-logic-level reasoning is established by Viper.

The automation technique described in this section is not complete, due to the heuristic choices made (particularly for selecting our dirty regions). As we show in the next section, it nonetheless works extremely well in practice (and accurately identifies bugs in faulty algorithms).

4.5 An Example Proof in our Frontend

To illustrate the benefits of our proof technique and automation, we list here the code and annotations needed to verify the PIP example. We only show the annotations that need be provided by an end-user, all the remaining parts of the proof are determined by our proof technique and translation to Viper.

Since our target verifier is Viper, which supports heap-dependent functions, we directly write field expressions to refer to field values of nodes x to which we have permission (denoted `acc(`x`)`). We use the Gr predicate from §4.1.1 to express our specifications: for instance, the assertion Gr(`X`, `nodeInv(_, X)`, `true`) denotes the flow graph of a set of nodes `X`, where each node `x` satisfies the node-local predicate `nodeInv(x, X)`, and the flow interface of `X` satisfies the trivial condition `true`. The `nodeInvIntermediate` predicate is used to denote states where the priority of a node has not yet been updated to match its flow.

We note that the annotations required from the end-user are minimal, and consist of flow domain definitions, local invariants on nodes' fields and flows, and annotations `_make_dirty` to manually add neighbours to the dirty region in certain cases.

```
1  // ---- Flow Domain definitions and PIP invariant:
2
3  type FlowDom = Multiset[Int]
4    // Contains:
5    // fd : Multiset[Int] → FlowDom
6    // fdZero : () → FlowDom
7
8  // The fields of a node:
9  field parent: Ref
10 field current_prio: Int
11 field default_prio: Int
12 field priorities: Multiset[Int]
```

```
13
14 // Definition of a node and good condition:
15 function edges(x: Ref, y: Ref) : Map[FlowDom, FlowDom]
16   requires acc(x)
17 {
18   (x != y ∧ x.parent != null ∧ x.parent == y) ?
19     {_ ↣ fd(Multiset(x.current_prio))} : {_ ↣ fdZero()}
20 }
21
22 function correct_priorities(x: Ref, from: Int, to: Int): Multiset[Int]
23   requires acc(x)
24   ensures result == (to > 0 ? x.priorities ∪ Multiset(to) : x.priorities) \ Multiset(from)
25
26 define nodeInvIntermediate(x, y, from, to, X)
27   // priorities are strictly positive
28     x.default_prio > 0 ∧ (∀ i: Int :: (i ∈ x.priorities) > 0 ==> i > 0)
29   // local priorities are consistent with inflow
30   ∧ fd(y == x ? correct_priorities(x, from, to) : x.priorities) == x.intf^in(x)
31   // x.current_prio is maximal priority of x and predecessors
32   ∧ x.current_prio == max(ms_max(x.priorities), x.default_prio)
33   // No self loops
34   ∧ x.parent != x
35   // Data structure is closed
36   ∧ x.parent != null ==> x.parent ∈ X
37
38 define nodeInv(x, X) nodeInvIntermediate(x, null, 0, 0, X)
39
40 // ---- PIP algorithm:
41
42 method updatePriorities(this: Ref, from: Int, to: Int, X: Set[Ref])
43   requires Gr(X, nodeInvIntermediate(_, this, from, to, X), true)
44   requires this ∈ X
45   ensures Gr(X, nodeInv(_, X), true)
46 {
47   if (from != to) {
48     var old_prio: Int := this.current_prio
49     if (to > 0) {
50       this.priorities := this.priorities union Multiset(to)
51     }
52     this.priorities := this.priorities setminus Multiset(from)
53     this.current_prio := max(ms_max(this.priorities), this.default_prio)
54     _make_dirty(this.parent)
55
56     if (this.current_prio != old_prio && this.parent != null) {
57       updatePriorities(this.parent, old_prio, this.current_prio, X)
58     }
59   }
60 }
61
62 method aquire(this: Ref, r: Ref, X: Set[Ref])
63   requires Gr(X, nodeInv(_, X), true)
```

```
  requires this ∈ X ∗ r ∈ X ∗ this != r ∗ this.parent == null
  ensures Gr(X, nodeInv(_, X), true)
{
  if (r.parent == null) {
    r.parent := this
    _make_dirty(r.parent)
    updatePriorities(this, 0, r.current_prio, X)
  } else {
    this.parent := r
    _make_dirty(this.parent)
    updatePriorities(r, 0, this.current_prio, X)
  }
}

method release(this: Ref, r: Ref, X: Set[Ref])
  requires Gr(X, nodeInv(_, X), true)
  requires this ∈ X ∗ r ∈ X
  ensures Gr(X, nodeInv(_, X), true)
{
  if (r.parent == this) {
    r.parent := null
    _make_dirty(this)
    updatePriorities(this, r.current_prio, 0, X)
  }
}
```

4.6 Evaluation

We evaluate the flow framework and our accompanying automation techniques for its proofs on a collection of challenging data-structure examples. These are hand-encoded into Viper, but our encodings simulate the systematic behaviour of a potential future front-end tool for this reasoning, requiring as input only the flow domain instantiation and specified code as described in §4.4.

Our collection of examples is presented in Table 4.1. Some examples (such as the PIP and Composite) concern edge-local flow domains, while most employ effectively-acyclic graphs (and the corresponding additional requirements on interface composition) to structure the reasoning and automation. We employ a variety of flow domains to succinctly capture the graph properties in question, and show the preservation of the key invariants of each data structure.

We chose our benchmarks based on whether at least one existing SL-based reasoning technique would struggle to cope with them (with the exception of the singly and doubly linked-list examples, which can be handled by most techniques). We next describe each of the non-trivial examples in detail:

Overlaid Data Structures The first non-trivial group of examples (Harris list, threaded tree) involves operations on overlaid data structures (traversal, insertion, deletion, etc.). The Harris list (see §4.2 for more details) is a singly-linked list that is arbitrarily overlayed with a *free* list

Table 4.1: The results of our evaluation. Here, "EL" means the flow domain is edge-local, while "EA" means the example employs effectively-acyclic graphs. The "buggy" variants are adapted from the correct code by intentionally seeding mistakes. All examples behave as expected (correct versions verify, buggy versions identify the correct errors). All timings were gathered on an Intel Core i7-7700K 4.20Ghz machine running (64 bit) Windows 10; timings were taken 7 times, the lowest and highest times discarded, and the remaining 5 averaged and reported to two decimal places (we observed no significant variations in timings)

Example Description	Properties Verified	Flow Domain Values	Class EL	Class EA	Variant	Average Time (secs)
EA Singly-linked-list (insert,delete,etc.)	exactly one path from hd to each node	Natural numbers (path-counting)		✓	correct	12.31
					buggy	10.62
EL Doubly-linked-list (insert,delete,etc.)	next vs. prev fields inverses	Multisets of references	✓		correct	12.75
					buggy	14.92
EA Doubly-linked-list (insert,delete,etc.)	next vs. prev; connected list	Multisets of references		✓	correct	28.27
					buggy	28.16
Harris List (insert, delete)	nodes in two overlaid lists; those on one are marked	Pairs of natural numbers		✓	correct	12.81
					buggy	12.02
Threaded Tree (traversal, insert)	nodes in tree overlaid with list	Pairs of natural numbers		✓	correct	40.54
					buggy	21.40
B-Tree	functional correctness of node level ops.	Maps of keys to Natural numbers		✓	correct	13.76
					buggy	15.10
Hash Table	functional correctness of node level ops.	Maps of keys to Natural numbers		✓	correct	12.59
					buggy	13.63
Composite Pattern	Node values store subtree size	Integers	✓		correct	10.22
					buggy	10.45
Subject Observer Pattern	inverse reference invariants	Multisets of references		✓	correct	19.34
					buggy	48.94
Priority Inheritance Protocol (PIP)	Priorities accurate; no priority inversion	Multisets of integers	✓		correct	35.55
					buggy	35.93
Speculative Dijkstra Shortest Path	functional correctness: computes shortest paths	Multisets of costs ($\mathbb{N} \cup \{\infty\}$)	✓		correct	18.69
					buggy	75.86

consisting of marked nodes (inspired by the drain technique of manual memory management). The threaded tree is a commonly-used data structure, used, for example, in the Linux deadline I/O scheduler, that facilitates two types of access to elements: FIFO access via the linked list and efficient key-based lookup via the tree. Its structure is therefore a linked list overlaid arbitrarily over a binary tree. We consider a simplifed version of the algorithm that traverses down the tree to a nondeterministic leaf and inserts a new node into both the three and the list. These examples are generally difficult to handle with existing techniques because such data structures allow multiple traversal strategies (following different sets of pointers in the heap) and the invariants of the overlays can be intertwined (e.g., every node is contained in both structures).

Using flows, we find that compared to the singly-linked list benchmarks, the specification effort is of similar complexity and scales-up in size roughly linearly with the number of overlays. In particular, the proof of a traversal of the Harris list looks almost identical to that of a simple

singly-linked list.

Search Structures The next group of benchmarks (B-tree, hash table) is inspired by the technique for verifying functional correctness of search structures presented in [Krishna et al. 2018b, §7]. The specifications of these examples are challenging because they involve non-trivial combinations of data and structural invariants. For each of these complex structures, we verify functional correctness of the node-level operations such as member, insert, and delete.

As we will see in Chapter 5, using flows we can break down the correctness of concurrent search structures into a concurrent proof of template algorithms that manage interference between threads and find the correct node on which to operate, and a sequential proof of node-level operations on the appropriate node. The proofs we perform satisfy the specifications required to be part of such a modular proof, and thus contribute towards the proof of the complete concurrent algorithms. This shows that the flow framework and our automation technique can scale to handle complex real-world concurrent algorithms, some of which (B-link tree) have not been mechanically verified before. For more details, see Chapter 5.

OOP design patterns Next, we consider object-oriented programming (OOP) design patterns (Composite, Subject/Observer). Our example of the Composite design pattern is the one studied by Summers and Drossopoulou [2010] and involves a tree where each node n stores the size of the subtree rooted at n. When one adds a subtree to a node n in the tree, the algorithm has to add the size of the subtree to the value stored in n, and then recursively traverse up n's ancestors and correct their size fields.

The Subject/Observer example [Parkinson 2007] is a program consisting of a subject, which contains a value of interest, and several observers, who have a cached copy of the value. When new observers are created, they register themselves with the subject. When the subject changes its value, it iterates through the observers it has in its register, and notifies them of the new value so that they can update their caches. The invariant is that the observers' caches match the value stored by the subject.

Both these algortihms involve multi-object invariants which are known to be challenging to handle at a per-node level of granularity in separation logics. In a sense, using flows we can mimic existing proof strategies for such examples based on object invariants [Summers and Drossopoulou 2010] within separation logic, obtaining the additional benefits of modular framing.

Graph Algorithms The last two examples deal with recursive properties of general graphs. The PIP example is a complete version of the code shown in Fig. 3.1. The algorithm, and why its verification is challenging, was described in §1.1, and §4.5 contains a full code listing. The specification for the PIP states that the current priorities stored by each node is correct (i.e. the maximum of the priority of any node waiting on it), and thus that there are no priority inversions. The Dijkstra shortest path algorithm is a sequential variant of the concurrent algorithm considered by Raad et al. [2016]. We verify full functional correctness of this algorithm – that it correctly computes the length of the shortest path to every vertex.

We are not aware of any existing SL-based technique that can automatically verify these kinds of proofs. Moreover, we are not aware of any existing *local* proofs of the properties we considered for these benchmarks. We note that Sergey et al. [2015]; Raad et al. [2016] verify concurrent versions of such algorithms. These works focus on modular reasoning techniques for the concurrency aspects (which we ignore here) while the reasoning about the considered graph properties requires some non-local reasoning steps.

Buggy Versions We include both correct and buggy variants of the code, in order to demonstrate that we can effectively identify bugs in faulty proofs. Moreover, we aim to show that our automation is not significantly slower in searching for a wrong proof. We created the buggy versions of the code from the original correct versions by manually introducing common types of errors: off-by-one errors on array indices, failing to update array length variables after an array update, failing to check if a reference is null, mixing up local variables, arithmetic errors, etc.

As can be seen, the time taken to check each proof is reasonable despite the absence of manual intervention, and a variety of examples which go beyond the state-of-the-art for any proof technique known to enable a similar degree of automation.

4.7 Related Work

An abundance of SL variants provide complementary mechanisms for modular reasoning about programs (e.g. [Jung et al. 2018b; Raad et al. 2015; Sergey et al. 2015]). Most have in common that they are parameterized by the underlying separation algebra; our flow-based reasoning technique can be easily integrated into these existing logics.

Recursive data structures are classically handled in SL using *recursive predicates* [O'Hearn et al. 2001; Reynolds 2002]. There is a rich line of work in automating such reasoning within decidable fragments (e.g. [Berdine et al. 2004; Iosif et al. 2014; Katelaan et al. 2019; Piskac et al. 2013; Enea et al. 2017; Qiu and Wang 2019]). However, recursive definitions are problematic for handling e.g. graphs with cycles, sharing and unbounded indegree, overlaid structures and unconstrained traversals.

The most common approach to reason about irregular graph structures in SL is to use iterated separating conjunction [Yang 2001a; Müller et al. 2016] and describe the graph as a set of nodes each of which satisfies some local invariant. This approach has the advantage of being able to naturally describe general graphs. However, it is hard to express non-local properties that involve some form of fixpoint computation over the graph structure. One approach is to abstract the program state as a mathematical graph using iterated separating conjunction and then express non-local invariants in terms of the abstract graph rather than the underlying program state [Hobor and Villard 2013; Sergey et al. 2015; Raad et al. 2016]. However, a proof that a modification to the state maintains a global invariant of the abstract graph must then often revert back to non-local and manual reasoning, involving complex inductive arguments about paths, transitive closure, and so on. Our technique and Viper encoding also exploit iterated separating conjunction for the underlying heap ownership, with the key benefit that flow interfaces exactly

capture the necessary conditions on a modified subgraph in order to compose with *any* context and preserve desired non-local invariants.

The flow framework presented in this chapter is inspired by the one presented in [Krishna et al. 2018b]. In addition to the technical innovations made here (general proof technique that integrates with existing SLs and proof automation), the most striking difference is in the underlying meta theory. The prior flow framework required flow domains to form a semiring; the analogue of edge functions are restricted to multiplication with a constant, which must come from the same flow value set. Our flow framework decouples the algebraic structure defining how flow is *aggregated* from the algebraic structure of the edge functions. As a consequence, we obtain a more general framework that applies to many more examples, and with simpler flow domains. Strictly speaking, the prior and our framework are incomparable as the prior did not require that flow aggregation is cancellative. As we argue in §3.1, cancellativity is a natural requirement for local reasoning, and is critical for ensuring that the inflow of a composed graph is uniquely determined. Instead of demanding cancellativity, Krishna et al. [2018b] require proofs to reason about flow interface *equivalence classes*. This complicates proofs (and their automation, introducing quantifier alternations), and entails strong restrictions on modifications of cyclic structures.

An alternative approach to using SL-style reasoning is to commit to global reasoning but remain within decidable logics to enable automation [Itzhaky et al. 2013; Klarlund and Schwartzbach 1993; Wies et al. 2011; Madhusudan et al. 2012; Lahiri and Qadeer 2008]. However, such logics are restricted to certain classes of graphs and certain types of properties. For instance, reasoning about reachability in unbounded graphs with two successors per node is undecidable [Immerman et al. 2004]. Recent work by Ter-Gabrielyan et al. [2020] shows how to deal with modular framing of *pairwise reachability* specifications in an imperative setting. Their framing notion has parallels to our notion of interface composition, but allows subgraphs to *change* the paths visible to their context. The work is specific to a reachability relation, and cannot express the rich variety of custom graph properties available by instantiating flow domains in our technique.

Scope and Limitations of the Flow Framework The universal flow (Definition 3.22) shows that flows are powerful enough to capture any graph property. This means that flow-based proofs must make a trade-off between how global the invariants they reason about are and how local the reasoning about the program is. If a flow captures an invariant that is intrinsically global, then the kinds of modifications permitted by flow interfaces will also have large flow footprints. While the theory of flows permits reasoning about such large/unbounded dirty regions, automating such proofs will be hard unless the footprint happens to have well-defined structure. However, this is in some sense inescapable, and in such cases the algorithm will also have to repair the invariant of all nodes in the flow footprint. Thus, techniques such as the edge-local transformation can help obtain a local proof that mirrors the actions of the algorithm.

We have not yet formulated a precise relation between inductive predicates and flows, but this is an interesting question to explore in future work. However, the programs and data structures we have studied so far suggest that anything one can define using an inductive predicate can also be defined using flows. Since any graph property can be captured by flows, the fact that the flow equation is a fixpoint equation that mirrors the inductive unrolling of inductive definitions

suggests that one can achieve similar abstraction capabilities using flows. It would be interesting to formulate a precise transformation from an inductive predicate definition to a flow domain and local invariant.

4.8 Conclusion

This chapter has shown how two of the key classes of the flow framework (edge-local flow domains and effectively-acyclic graphs) can be built upon to provide automated, simple proof checking for a wide variety of challenging examples.

5 Concurrent Search Structure Templates

This chapter presents a major application of the flow framework to simplifying and modularizing the verification of a large class of concurrent data structures: search structures. We propose a methodology for verifying template algorithms for concurrent search structures that enables proofs to be compositional in terms of program structure and state, and exploit both thread and algorithmic modularity. Using this method, we mechanically prove several complex real-world data structures such as the B-link tree that are beyond the capability of existing techniques for mechanized or automated formal proofs.

Our proofs are much simpler and more automated than prior (pencil-and-paper) proofs of comparable structures. Further, the template-based modularity in our proofs allow us to mix and match synchronization protocols and heap representations with negligible additional proof effort. For example, most of the core operations on B-trees such as insertion and deletion of a key into a node are shared between the B-link (which uses the link technique) and the B+ tree (which uses the give-up technique). Hence, they need to be verified only once.

We begin in §5.1 by describing the B-link tree and use it to introduce the idea of a template algorithm. This idea of viewing search structure algorithms abstractly as template algorithms on graphs is due to Shasha and Goodman [1988]. Our contribution is to formalize their ideas and in §5.2 we describe our proof technique in detail, which relies on two important steps. First, §5.2.1 shows how to encode the edgeset framework using flows. Second, we embed flow interfaces as ghost state in Iris in §5.2.2, a combination that lets us prove that our template algorithms contextually refine their sequential specifications while abstracting from the implementation. This is demonstrated in §5.2.3, which puts this all together and verifies the link template algorithm. A summary of our verification effort is provided in §5.3. We perform the template proofs in Iris/Coq and verify the implementations in GRASShopper, in order to bring the proofs of highly complicated implementations such as B-link trees within reach. We close with a survey of related work in §5.4 and conclude in §5.5.

5.1 Overview

This section motivates and demonstrates our approach using the B-link tree implementation of a search structure and the link template algorithm that generalizes it. A search structure is

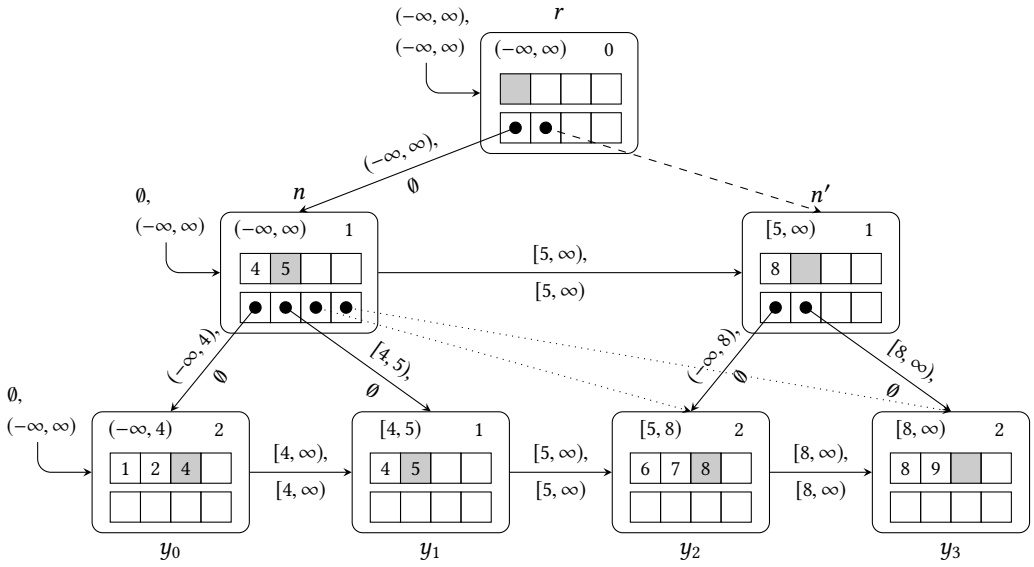


Figure 5.1: An example B-link tree state in the middle of of a split. Node n was full, and has been half-split and children y_2 and y_3 have been transferred to new node n' (old edges are shown as dotted lines), but the complete split has yet to add n' to the parent r (the dashed edge). Each node shows the array of keys in the top left, the array of pointers in the bottom left, l (number of keys) in the top right, and its inflow (see §3.2.1) in the top left. The key in the gray box is not considered part of the contents and determines the edgeset of the link edge. The edges are labelled with edgesets and linksets (see §5.2.1). The global inflow is shown as curved arrows on the top left of nodes, and is omitted when zero.

a key-based store that implements three basic operations: search, insert, and delete. 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. For simplicity, the presentation here treats search structures as containing only keys (i.e. as implementations of mathematical sets), but all our proofs can be easily extended to consider search structures that store key-value pairs.

5.1.1 B-link Trees

The B-link tree (Figure 5.1) is an implementation of a concurrent search structure based on the B-tree. A B-tree is a generalization of a binary search tree, in that a node can have more than two children. In a binary search tree, each node contains a key k_0 and up to two pointers y_l and y_r. An operation on k takes the left branch if $k < k_0$ and the right branch otherwise. A B-tree generalizes this by having l sorted keys $k_0, \ldots, k_{l-1}$ and $l + 1$ pointers $y_0, \ldots, y_l$ at each node, such that $B \leq l + 1 < 2B$ for some constant B. At internal nodes, an operation on k takes the branch y_i if $k_{i-1} \leq k < k_i$. Only the keys stored in leaf nodes are considered the contents of a B-tree; internal nodes contain "separator" keys for the purpose of routing only. When an operation arrives at a leaf node n, it proceeds to insert, delete, or search for its query key in the keys of n. To avoid interference, each node has a lock that must be held by an operation before it reads from or writes to the node.

When a node n gets full, a separate maintenance thread performs a split operation by transferring half its keys (or pointers, if it is an internal node) into a new node n', and adding a link to n' from n's parent. In the concurrent setting, one needs to ensure that this operation does not cause concurrent operations at n looking for a key k that was transferred to n' to conclude that k is not in the structure. The B-link tree solves this problem by linking n to n' and store a key k' (the key in the gray box in the figure) that indicates to concurrent operations that all keys $k > k'$ can be reached by following the link edge. For efficiency, this split is performed in two steps: (i) a half-split step that locks n, transfers half the keys to n', and adds a link from n to n' and (ii) a complete-split performed by a separate thread that takes half-split nodes n, locks the parent of n, and adds a pointer to n'.

Figure 5.1 shows the state of a B-link tree where node y_2 has been fully split, and its parent n has been half split. The full split of y_2 moved keys $\{8, 9\}$ to a new node y_3, added a link edge, and added a pointer to y_3 to its (old) parent n. However, this caused n to become full, resulting in a half split that moved its children $\{y_2, y_3\}$ to a new node n' and added a link edge to n'. The key 5 in the gray box in n directs operations on keys $k \geq 5$ via the link edge to n'. The figure shows the state after this half split but before the complete-split when the pointer of n' will be added to r.

5.1.2 Abstracting Search Structures using Edgesets

The link technique is not restricted to B-trees: consider a hash table implemented as an array of pointers, where the ith entry points to a bucket node that contains an array of keys $k_0, \ldots, k_l$ that all hash to i. When a node n gets full, it is locked, its keys are moved to a new node n' with twice the capacity, and n is linked to n'. Again, a separate operation locks the main array entry and updates it from n to n'.

While these two data structures look completely different, the main operations of search, insert, and delete follow the same abstract algorithm. In both, there is some rule by which operations are routed from one node to the next, and both introduce link edges when keys are moved to ensure that no other operation loses its way.

To concretize this intuition, let the *edgeset* of an edge (n, n'), written $\text{es}(n, n')$, be the set of query keys for which an operation arriving at a node n traverses (n, n'). For the B-link tree in Figure 5.1, the edgeset of (n, y_1) is $[4, 5)$ and the edgeset of the link edge (y_0, y_1) is $[5, \infty)$. Note that 4 is in the edgeset of (y_0, y_1) even though an operation on 4 would not normally reach y_0; in order to make edgeset a local quantity, we say $k \in \text{es}(n, n')$ if an operation on k would traverse (n, n') assuming it somehow found itself at n. In the hash table, assuming there exists a global root node, the edgeset from the root to the ith array entry is $\{k \mid hash(k) = i\}$. The edgeset from an array entry to the bucket node is the set of all keys KS, as is the edgeset from a deleted bucket node to its replacement.

5.1.3 The Link Template Algorithm

Figure 5.2 lists the link template algorithm [Shasha and Goodman 1988] that uses edgesets to describe the algorithm used by all core operations for both B-link trees and hash tables in a

```
1 let rec traverse n k =
2   lockNode n;
3   match findNext n k with
4   | None -> n
5   | Some n' -> unlockNode n;
6       traverse n' k
```

```
7 let rec searchStrOp ω r k =
8   let n = traverse r k in
9   match decisiveOp ω n k with
10  | None -> unlockNode n;
11      searchStrOp ω r k
12  | Some res -> unlockNode n; res
```

Figure 5.2: The link template algorithm, which can be instantiated to the B-link tree algorithm by providing implementations of helper functions `findNext` and `decisiveOp`. `findNext n k` returns `Some n'` if $k \in \mathrm{es}(n, n')$ and `None` if there exists no such n'. `decisiveOp n k` performs the operation ω (either search, insert, or delete) on k at node n.

uniform manner. The algorithm assumes that an implementation provides certain primitives or helper functions, such as `findNext` that finds the next node to visit given a current node n and a query key k, by looking for an edge (n, n') with $k \in \mathrm{es}(n, n')$. For the B-link tree, `findNext` does a binary search on the keys to find the appropriate pointer to follow, while for the hash table, when at the root it returns the edge to the array element indexed by the hash of the key, and at bucket nodes it follows the link edge if it exists. The function `searchStrOp` can be used to build implementations of all three search structure operations by implementing the helper function `decisiveOp` to perform the desired operation (read, add, or remove) of key k on the node n.

An operation on key k starts at the root r, and calls a function `traverse` on line 8 to find the node on which it should operate. `traverse` is a recursive function that works by following edges whose edgesets contain k (using the helper function `findNext` on line 3) until the operation reaches a node n with no outgoing edge having an edgeset containing k. Note that the operation locks a node only during the call to `findNext`, and holds no locks when moving between nodes. `traverse` terminates when `findNext` does not find any n' such that $k \in \mathrm{es}(n, n')$, which, in the B-link tree case means it has found the correct leaf to operate on. At this point, the thread performs the decisive operation on n (line 9). If the operation succeeds, then `decisiveOp` returns `Some res` and the algorithm unlocks n and returns `res`. In case of failure (say an insert operation encountered a full node), the algorithm unlocks n, gives up, and starts from the root again.

If we can verify this link template algorithm with a proof that is parametrized by the helper functions, then we can reuse the proof across diverse implementations.

5.1.4 A Proof Strategy for Template Search Structures

As the link template algorithm is parametrized by the concrete data structure, its proof cannot use any data-structure-specific invariants (such as that the array of keys in a B-tree is sorted). The edgeset framework provides a correctness condition for search structure algorithms in terms of reachability properties of sets of keys on a mathematical graph, abstracting from the data layout of the implementation.

Let the contents of a node be the set of keys that are stored at that node (for the B-link tree in Figure 5.1 the contents of y_0 are $\{1, 2\}$, while the contents of internal nodes like n are $\emptyset$). We let the state of a data structure be the graph whose edges are labelled with edgesets and nodes

with their contents. The abstract state of a graph is then the union of the contents of all its nodes. Proving that the link template refines its abstract specification requires us to prove that the decisive operation updates the abstract state appropriately. In our B-link tree example, say an operation seeking to delete 3 arrived at node y_0 and returned because 3 was not present, then the proof must show that 3 is not present anywhere else in the structure. Intuitively, we know that this is true because the rules defining a B-link tree ensure that y_0 is the only node where 3 can be present.

To generalize this argument to arbitrary search structures, we build on the concept of edgesets. 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, the pathset of the path between r and n' in Figure 5.1 is $(-\infty, \infty) \cap [5, \infty) = [5, \infty)$. With this, we define the *inset* of a node n, written $\mathsf{ins}(n)$, as the union of the pathsets of all paths from the root node to n (B-link trees may have several paths from the root to a given leaf node). Let the *outset* of n, $\mathsf{outs}(n)$, be the keys in the union of edgesets of edges leaving n. If we take the inset of a node n, and subtract the outset, we get the *keyset* of n, $\mathsf{ks}(n)$. Intuitively, the keyset of a node n is the set of keys that if present in the structure, must be in n. Coming back to our example, the keyset of node y_0 is $(-\infty, 4) \setminus [4, \infty) = (-\infty, 4)$, and so it suffices for the delete operation to ensure that 3 is not present in y_0.

We enforce the above interpretation of the keyset using the following *good state* conditions:

(GS1) The contents of every node are a subset of the keyset of that node.

(GS2) The edgesets of two distinct edges leaving a node are disjoint.

For data structures with a single root, (GS2) ensures that the keysets of two distinct nodes are disjoint. This, along with (GS1), tells us that we can treat the keyset of n as the set of keys that n can potentially contain. In good states, 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. Given a good state, if an operation looks for, inserts, or deletes k at a node n such that k is in the keyset of a node n, then the keyset theorem of Shasha and Goodman [1988] shows that the operation modifies the abstract state correctly.

How does the link template ensure that $k \in \mathsf{ks}(n)$ when `decisiveOp` is called? In the absence of split operations and link edges, this follows because we start off at the root r, where by definition $k \in \mathsf{ins}(r)$, and traverse an edge (n, n') only when $k \in \mathsf{es}(n, n')$, maintaining the invariant that $k \in \mathsf{ins}(n)$. When there does not exist an outgoing edge with k in the edgeset, we know by definition that $k \in \mathsf{ks}(n)$.

In the presence of split operations, this invariant breaks down because the inset of a node n shrinks after a split, so that k might have been in the $\mathsf{ins}(n)$ before the split but not afterwards. Note, however, that if one traverses the link edge, one can get back to a node with k in its inset. The way to formalize a more general invariant is to define the *inreach* of a node n as

$$\mathsf{inr}(n) := \mathsf{ins}(n) \cup \bigcup_{n'} \mathsf{es}(n, n') \cap \mathsf{inr}(n').$$

Intuitively, $\mathsf{inr}(n)$ is the set of keys k for which if we follow edges labelled with k from n then we will eventually reach a node n' with $k \in \mathsf{ins}(n')$. For example, in Figure 5.1 the inreach of y_1 is $[4, \infty)$ even though its inset is only $[4, 5)$, for it can reach the nodes with k in their inset for all $k \geq 4$ by following link edges. The invariant of the traversal is then that $k \in \mathsf{inr}(n)$. This is true at the root, because $\mathsf{inr}(r) = \mathsf{ins}(r) = \mathsf{KS}$, and it is preserved during the traversal even with concurrent splits. When `findNext` returns `None`, the definition of inreach implies that $k \in \mathsf{inr}(n) \setminus \mathsf{outs}(n) \subseteq \mathsf{ks}(n)$, which by the keyset theorem gives us correctness of the decisive operation.

The edgeset framework and keyset theorem thus give us abstract conditions under which a template algorithm is correct. However, reasoning about insets and inreach is still challenging, because they are global inductively-defined quantities of the data structure. If we can write local pre- and post-conditions for helper functions such as `decisiveOp`, then the proof of an implementation can reason only about the node that the helper function modifies. In the next section, we show how to reason about the correctness conditions for template algorithms that rely on global quantities using local reasoning.

5.2 Verifying Search Structure Templates

This section shows how to tie together the edgeset framework and flow interfaces in Iris in order to verify template algorithms for concurrent search structures. We do this using the proof of the link template from §5.1 as an example. The other template algorithms we prove, as well as the implementations we consider, are described in the next section. A formal introduction to Iris and the underlying programming language semantics is, unfortunately, beyond the scope of this thesis. We provide intuition for the key logical constructs and reasoning steps as and when they are used; for a more detailed introduction to Iris see [Jung et al. 2018a].

Proving correctness of data structures generally involves showing memory safety and functional correctness. In this work, we prove that template algorithms such as the link template satisfy a specification that encapsulates correctness as well as safety: contextual refinement. An implementation program e_1 contextually refines a specification program e_2 if and only if, for every possible client, each behavior when using e_1 is a possible behavior of e_2.

To prove contextual refinement, we use ReLoC [Frumin et al. 2018], an extension of Iris with first-class support for reasoning about refinement. However, to simplify the presentation, we show only the Hoare-style proof of the invariant needed for refinement in this section and indicate at the appropriate points what extra proof obligations are needed for the full refinement proof. All free variables in the intermediate assertions are implicitly existentially quantified.

Let us abstractly represent the state of a search structure as the set of keys C that it contains. We define the specification program `searchStrSpec` ω for search structure operation ω (either search, insert, or delete) on query key k as an atomic step that modifies the set C to a new set C', and returns the value res, such that the predicate $\Psi_\omega(k, C, C', \mathrm{res})$ defined in Figure 5.3 holds. Note that as our specification (a mathematical set ADT) is atomic, we can infer that our implementation is linearizable [Filipovic et al. 2009].

The high-level idea behind the refinement proof is to relate the state of the implementation to

$$\Psi_\omega(k, C, C', \text{res}) \coloneqq \begin{cases} C' = C \wedge (\text{res} \iff k \in C) & \omega = \texttt{search} \\ C' = C \cup \{k\} \wedge (\text{res} \iff k \notin C) & \omega = \texttt{insert} \\ C' = C \setminus \{k\} \wedge (\text{res} \iff k \in C) & \omega = \texttt{delete} \end{cases}$$

Figure 5.3: Abstract specification of search structure operations.

the state of the specification so that we can show that the implementation modifies the state in a manner consistent with the specification. We will do this in Iris using an invariant Inv which is a formula in Iris' logic that we formally define in §5.2.3 but describe intuitively here. Since efficient concurrent implementations usually distribute the state over a set of nodes, a natural first step is to associate each node with the set of keys it contains and use the invariant Inv to express that the union of contents of all nodes is equal to the abstract state C. However, as mentioned in §5.1.4, this is not enough if one wants to do local reasoning. For instance, if we want to prove that a delete operation on k that removed k from node n is correct, we additionally need to show that k is not present in the contents of any other node in the structure[1].

The edgeset framework comes to our rescue here: if the implementation state satisfies the good state conditions of §5.1.4 and k is in the keyset of n, then we know that k cannot be present in any other node. Thus, Inv will also enforce that the state is a good state. As we have seen, to reason about the good state conditions locally, we encode them using flows. We will use ghost state (see §5.2.2) to keep track of the flow interface of each node, and use the node-local constraint γ to enforce that each node satisfies local versions of the good state conditions. Our invariant Inv will tie the flow interface of each node to the actual heap representation of the node. The flow interfaces also keep track of each node's contents using the node labels, and Inv will further enforce that the node label of the global interface should be equal to the abstract contents C.

Apart from showing that all threads maintain the invariant, we must additionally show that any assumptions made by a thread is not violated by the actions of other threads. For example, as discussed in §5.1.4, `traverse` relies on the fact that $k \in \mathsf{inr}(n)$ when it is called in order to guarantee that $k \in \mathsf{ks}(n)$ when it returns. So we must ensure that no other operation modifies the state in a way that violates $k \in \mathsf{inr}(n)$, which we can do by proving that no operation decreases the inreach of any node. One can think of this as being the protocol obeyed by each thread in the link technique.

The rest of this section explains how to implement this high-level proof structure in Iris. First, §5.2.1 explains how we use flow interfaces to enforce the edgeset framework's good state conditions and lift a proof that an operation correctly updated a node to a proof that the operation correctly updated the entire data structure. §5.2.2 then describes the resource algebras that we use to encode both flow interfaces and other ghost information needed for our proof, as well as to enforce the protocol by which the shared state is updated. Finally, in §5.2.3 we define the invariant Inv in Iris and prove that the link template algorithm maintains Inv. We also describe

[1]The essential property we require is that the contents of distinct nodes be disjoint, but disjointness is not a local condition either.

how Inv is strong enough to extend this proof to show refinement.

5.2.1 Encoding the Edgeset Framework using Flows

Our first task is to provide an encoding of the edgeset framework using flows that enables us to lift a proof that an operation correctly updated the contents of a single node in the search structure to a proof that it correctly updated the contents of the data structure as a whole.

To encode the edgeset framework using flows, we cannot use sets of keys as the flow domain because set union is not cancellative. Instead, we use multisets as our flow domain, as multiset union is cancellative and suffices to encode the invariants of the edgeset framework. Our flow domain is thus $(\mathbb{N}^{\mathsf{KS}}, \emptyset, \cup, E_{\mathsf{KS}})$, where we represent multisets as functions from keys to natural numbers (the number of times each key occurs), write $\cup$ for multiset union (pointwise addition of occurrence counts), and denote by E_{KS} the set of edge functions $e_K := (\lambda K'.\ \{k \rightarrowtail \min(K(k), K'(k))\})$ for $K \subseteq \mathsf{KS}$. We label each edge (n, n') in graph G by the function $e_{\mathsf{es}(n,n')}$, which encodes intersection by the edgeset $\mathsf{es}(n, n')$. We use a global inflow $in = (\lambda n.\ \{k \rightarrowtail (n = r\ ?\ 1 : 0)\})$, which demands that the searches for all keys k in the global graph start at the root r. The flow will then tell us for every node n and key k, how many paths there are to the node n that a search for k may follow. In particular, we have $\mathsf{flow}(n)(k) > 0$ iff k is in the inset of n.

We express the good state condition (GS1) by saying that every key k in n's contents is in the inset ($\mathsf{flow}(n)(k) > 0$), and there is no edge to n' with k in its edgeset. Similarly, we can express (GS2) by saying for any key k and other nodes $n_1 \neq n_2$, k is not in the edgeset of at least one of the two edges from n to n_1, n_2.

It is hard to define the inreach directly as a flow [Krishna et al. 2018a], so we encode an under-approximation of inreach that is sufficient for correctness. The key idea is to view the graph as an overlay of two structures: a standard structure where the flow computes the inset, and a link structure consisting only of the link edges. For the B-link tree, the main structure consists of the tree edges from nodes to their children, while the link structure is composed of one list per level. This is modeled in the flow framework by using the product of two key count domains (see Example 3.4) as the flow domain, where the first component calculates the inset as described above. The roots of the second component are the first nodes on each level (as shown in Figure 5.1), and the resulting flow at each node n is called the *linkset* of n, denoted $\mathsf{lnks}(n)$. The linkset of y_0 is $(-\infty, \infty)$ as it is the first leaf, and the linkset of y_2 is $[5, \infty)$. One can think of the linkset component as describing how keys are routed when they traverse link edges.

Note that in the B-link tree, the linkset happens to be equal to the inreach. In general, we require only that the linkset approximate the inreach in such a way that it has the following properties: First, if $k \in \mathsf{lnks}(n) \setminus \mathsf{ins}(n)$ then for every edge (n, n'), k is in the edge label of the inset component (i.e. the edgeset) of (n, n') if and only if k is in the linkset component of the edge label of (n, n'). This is used to prove that if $k \in \mathsf{ins}(n) \cup \mathsf{lnks}(n)$ and `findNext n k` returns `Some n'`, then $k \in \mathsf{ins}(n') \cup \mathsf{lnks}(n')$ (needed by the recursive call to `traverse`). Second, if $k \in \mathsf{lnks}(n) \setminus \mathsf{outs}(n)$, then $k \in \mathsf{ins}(n)$(which in turn implies $k \in \mathsf{ks}(n)$). We use this to infer that when `findNext` fails, we have found the right node n (k is in n or nowhere in the structure). These two properties mean that instead of the inreach it is sufficient to work with the inreach-approximation $\mathsf{ins}(n) \cup \mathsf{lnks}(n)$, which for simplicity we shall call inreach in the following. We enforce these properties in γ, the

local good condition on nodes.

Before describing γ, we introduce some shorthand notation for clarity (these overload the symbols used when describing the edgeset framework because they express the same quantities):

$$\mathsf{ins}(I,n) \coloneqq \{k \mid I^{in}(n)_{\mathsf{is}}(k) \geq 1\} \qquad \mathsf{lnks}(I,n) \coloneqq \{k \mid I^{in}(n)_{\mathsf{ls}}(k) \geq 1\}$$
$$\mathsf{outs}(I) \coloneqq \{k \mid \exists n'.\ I^{out}(n')_{\mathsf{is}}(k) \geq 1\} \qquad \mathsf{es}(I,n,n') \coloneqq \{k \mid \mathsf{dom}(I) = \{n\} \wedge I^{out}(n')_{\mathsf{is}}(k) \geq 1\}$$

where m_{is} and m_{ls} denote the inset and linkset component of the flow value m.

We assume that the implementation uses two ghost fields to store the contents and the inreach-approximation of each node, and use two logical variables cn and inr that map nodes to their contents and inreach respectively. We enforce this in the node-local good condition γ, along with the good state conditions (GS1) and (GS2) and the properties needed for the desired interpretation of linkset:

$$\gamma(n, \vec{\mathsf{f}}_n, \_, J, \mathsf{cn}, \mathsf{inr}) \coloneqq \mathsf{cn}(n) \subseteq \mathsf{ins}(J_n, n) \setminus \mathsf{outs}(J_n) \tag{5.1}$$
$$\wedge\ (\forall n', n''.\ n' = n'' \vee \mathsf{es}(J_n, n, n') \cap \mathsf{es}(J_n, n, n'') = \emptyset) \tag{5.2}$$
$$\wedge\ (\forall k, n'.\ k \in \mathsf{inr}(n) \setminus \mathsf{ins}(J_n, n) \Rightarrow {J_n}^f(n, n')_{\mathsf{ls}}(k) = {J_n}^f(n, n')_{\mathsf{is}}(k)) \tag{5.3}$$
$$\wedge\ \mathsf{lnks}(J_n, n) \subseteq \mathsf{ins}(J_n, n) \cup \mathsf{outs}(J_n, n) \tag{5.4}$$
$$\wedge\ \vec{\mathsf{f}}_n = \{\mathtt{cont}\colon \mathsf{cn}(n), \mathtt{inr}\colon \mathsf{inr}(n), \dots\} \tag{5.5}$$

Here, conditions (5.1) and (5.2) encode the good state conditions (GS1) and (GS2). (5.3) and (5.4) are the two constraints on the linkset that we described earlier. Finally, (5.5) ties the logical maps cn, inr to the ghost field values.

We also require the following constraints on the global interface:

$$\varphi(I) \coloneqq (\forall n, k. I^{in}(n)_{\mathsf{is}}(k) = (n = r\ ?\ 1 : 0)) \wedge I^{out} = \lambda_0$$

This says that in the inset flow domain component, the global inflow assigns a key count of 1 to the root r, and 0 for every other node, for all keys (i.e. all searches start at the root). It does not restrict the global inflow in the linkset component. We also require that the global interface is closed (i.e. has no outgoing flow).

The good condition γ and the global interface constraint φ together result in a flow that computes the inset and the linkset of each node. They also enforce the good state conditions of the edgeset framework.

Finally, we can use these quantities and constraints to formulate a version of the Keyset Theorem from [Shasha and Goodman 1988], which reduces the problem of proving the search structure specification Ψ_ω for the global region X to proving it for the single modified node n:

KEYSET-THM
$$\frac{\begin{array}{ll}\forall x.\ \gamma(x, \_, \_, J, \mathsf{cn}, \_) & n \in X \qquad \varphi(J_X) \\ \forall x \neq n.\ \mathsf{cn}(x) = \mathsf{cn}'(x) & k \in \mathsf{ins}(J_n, n) \setminus \mathsf{outs}(J_n)\end{array}}{\Psi_\omega(k, \mathsf{cn}(n), \mathsf{cn}'(n), \mathsf{res}) \Rightarrow \Psi_\omega(k, \mathsf{cn}(X), \mathsf{cn}'(X), \mathsf{res})}$$

Here, we lift the contents map to a set of nodes as $\mathsf{cn}(X) \coloneqq \bigcup_{x \in X} \mathsf{cn}(x)$. This lemma is a pure assertion about flow interfaces and can be proven by induction on the number of nodes in X.

5.2.2 Resource Algebras and Ghost State

Iris models both the knowledge of threads about the shared state (e.g. $k \in \mathsf{ks}(n)$) and protocols for modifying the shared state (e.g. inreach cannot decrease) using the notion of *ghost state*. Ghost state, also known as logical or auxiliary state, is a type of primitive resource (analogous to the points-to predicate from standard separation logics) that helps with the proof but has no effect on run-time behavior. Ghost state can be allocated by the prover at any time at unused *ghost names*, the analogue of memory addresses for concrete locations, and will contain values drawn from a user-specified resource algebra (RA). Iris expresses ownership of ghost state by the proposition $\boxed{a}^{\gamma}$ which asserts that ownership of a piece $a \in M$ of the ghost location γ. Ghost state can be split and combined according to the rules of the underlying RA: $\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$. Furthermore, Iris maintains the invariant that the composition of all the pieces of ghost state at a particular location is valid (in terms of the $\overline{\mathcal{V}}$ predicate from Figure 2.1).

For instance, consider the authoritative RA that we will use to keep track of the inreach of each node. Given an RA M, the authoritative RA $\textsc{Auth}(M)$ (see [Jung et al. 2018a] for the formal definition) can be used to model situations where one party owns the *authoritative* element $a \in M$ and other parties are allowed to own fragments $b \in M$, with the invariant that all fragments $b \preccurlyeq a$. This can be used to model, for example, a shared heap, where there is a single authoritative heap a and each thread owns a fragment of it. The invariant that all fragments $b \preccurlyeq a$ implies that the fragments owned by all threads are consistent. We write $\bullet\, a$ for ownership of the authoritative element and $\circ\, b$ for fragmental ownership.

We use an authoritative RA of sets of keys, at locations $\gamma_{i(n)}$ for each node n, to encode the inreach of each node. From the definition, one can show that this RA satisfies the following properties:

$$\textsc{auth-set-upd}\ \frac{X \subseteq Y}{\bullet X \rightsquigarrow \bullet Y} \qquad \textsc{auth-set-snapshot}\ \bullet X \rightsquigarrow \bullet X \cdot \circ X \qquad \textsc{auth-set-valid}\ \frac{\overline{\mathcal{V}}(\bullet X \cdot \circ Y)}{Y \subseteq X}$$

We store the inreach of a node n with interface I_n in the shared state as $\boxed{\bullet \mathsf{inr}(n)}^{\gamma_{i(n)}}$. Threads can take snapshots of the inreach of a node using auth-set-snapshot and move a fragment $\exists X.\, \boxed{\circ X}^{\gamma_{i(n)}}$ to their local state. This allows them to make assertions such as $\exists X.\, \boxed{\circ X}^{\gamma_{i(n)}} * k \in X$ for some key k, which in conjunction with auth-set-valid encodes the knowledge that $k \in \mathsf{inr}(n)$. This knowledge is stable under interference by other threads, for the only frame-preserving update permitted by this RA is auth-set-upd, which allows threads to increase the inreach of a node.

We also use an authoritative RA of flow interfaces $\textsc{Auth}(\mathsf{FI})$ to keep track of the flow interface ghost state of our algorithms. Using Theorem 3.18, we can show that this RA permits the following non-deterministic frame-preserving update (which is a generalization of the standard frame-preserving update presented in §2.2.1, for details see [Jung et al. 2018a]):

$$\textsc{auth-fi-upd}\ \frac{I_1 \precsim I_1'}{(\bullet\, I, \circ\, I_1) \rightsquigarrow \left\{(\bullet\, I', \circ\, I_1') \;\middle|\; I \precsim I' \wedge \exists I_2.\, I = I_1 \oplus I_2 \wedge I' = I_1' \oplus I_2\right\}}$$

$\{\hbar(n, I_n, \mathsf{cn}, \mathsf{inr})\}$
`findNext n k`
$\left\{v.\ \hbar(n, I_n, \mathsf{cn}, \mathsf{inr}) * (v = \mathsf{None} * k \notin \mathsf{outs}(I_n)) \right.$
$\left. \quad \vee\ v = \mathsf{Some}(n') * k \in \mathsf{es}(I_n, n, n')) \right\}$

$\{\hbar(n, I_n, \mathsf{cn}, \mathsf{inr}) * k \in \mathsf{ins}(I_n, n) * k \notin \mathsf{outs}(I_n)\}$
`decisiveOp` ω `n k`
$\left\{v.\ v = \mathsf{None} * \hbar(n, I_n, \mathsf{cn}, \mathsf{inr}) \right.$
$\left. \vee\ v = \mathsf{Some}(v') * \hbar(n, I'_n, \mathsf{cn}', \mathsf{inr}') * \Psi_\omega(k, \mathsf{cn}(n), \mathsf{cn}'(n), v') \right.$
$\left. \quad * I_n \precsim I'_n * \mathsf{inr}(I_n, n) = \mathsf{inr}(I'_n, n) \right\}$

Figure 5.4: Specifications of helper functions that are defined by implementations.

Finally, given any set S, it is easy to see that we have an RA $(2^S, \mathsf{True}, \lambda X.\emptyset, \cup)$, which permits a frame-preserving update $X \rightsquigarrow Y$ for any $X, Y \subseteq S$. We will use a set RA in our proof to keep track of the global contents, which is the state of the specification program.

5.2.3 Proof of the Link Template

Since Iris allows user-defined ghost state, we can simplify the proof technique and encoding from §4.1.1. First, we store the singleton interface of each node as a resource $\boxed{\circ I_n}^{\gamma_I}$ instead of using a ghost field `intf`. We also do not use the node and graph predicates N and Gr, and instead use a direct encoding into Iris to keep the specifications simple. Instead of an abstraction function edges, we assume an abstract *heap representation* predicate (whose definition is implementation-specific, and provided by the user for implementation proofs) $\hbar(n, I_n, \mathsf{cn}, \mathsf{inr})$ that must satisfy

$$\hbar(n, I_n, \mathsf{cn}, \mathsf{inr}) \Rightarrow n \mapsto \vec{\mathsf{f}}_n * \gamma(n, \vec{\mathsf{f}}_n, \_, \{n \rightarrowtail I_n\}, \mathsf{cn}, \mathsf{inr}).$$

Figure 5.5 presents a proof outline of the link template algorithm, where the intermediate assertions in braces show the context of the proof (the premises that are currently available). Our proof assumes that the implementation-specific helper functions satisfy the specifications shown in Figure 5.4. For instance, the specification of `findNext` expects to be given a node n satisfying $\hbar(n, I_n, \mathsf{cn}, \mathsf{inr})$, and returns None if k is not in the outset of n or else it returns $\mathsf{Some}(n')$ if k is in the edgeset $\mathsf{es}(I_n, n, n')$. Similarly, the specification of `decisiveOp` expects a node $\hbar(n, I_n, \mathsf{cn}, \mathsf{inr})$ such that k is in the keyset of n ($k \in \mathsf{ins}(I_n, n) * k \notin \mathsf{outs}(I_n)$). If `decisiveOp` returns None then it returns the node unchanged. On the other hand, if it returns $\mathsf{Some}(v')$ then the node is now $\hbar(n, I'_n, \mathsf{cn}', \mathsf{inr}')$, but the inreach is unchanged ($\mathsf{inr}(I_n, n) = \mathsf{inr}(I'_n, n)$), the new interface is a contextual extension of the previous interface ($I_n \precsim I'_n$), and the return value satisfies the search structure specification with respect to the old and new contents of the node n ($\Psi_\omega(k, \mathsf{cn}(n), \mathsf{cn}'(n), v')$).

We also use a standard lock module, similar to the one in [Frumin et al. 2018], adapted to a setting where each node n in our data structure has its own lock that protects the interface and heap representation of n, $\boxed{\circ I_n}^{\gamma_I} * \hbar(n, I_n, \mathsf{cn}, \mathsf{inr})$. We assume the specification of the lock and unlock methods shown at the top of Figure 5.5, but note that our Iris proofs prove this specification.

The Hoare-style specification for `searchStrOp` is simply

$$\left\{\boxed{\mathsf{Inv}}^{\mathcal{N}}\right\}\ \texttt{searchStrOp}\ \omega\ \texttt{r k}\ \left\{\boxed{\mathsf{Inv}}^{\mathcal{N}}\right\}.$$

However, the refinement proof will also require us to show that if we execute the specification program `searchStrSpec` at the linearization point, then `searchStrSpec` returns the same value as

```
(* Let inFP(n) ≔ ∃N. ⌜∘N⌝^{γ_f} ∗ n ∈ N ,  inInr(n,k) ≔ ∃R. ⌜∘R⌝^{γ_{i(n)}} ∗ k ∈ R *)

{ Inv^N ∗ inFP(n) }  lockNode n  { Inv^N ∗ Node(n, I_n, I_n) }
{ Inv^N ∗ Node(n, I_n, I_n) }  unlockNode n  { Inv^N }

{ Inv^N ∗ inFP(n) ∗ inInr(n,k) }
let rec traverse n k =
  lockNode n;
  { Inv^N ∗ Node(v, I_v, I_v) ∗ inFP(n) ∗ inInr(n,k) }
  match findNext n k with
  | None -> { Inv^N ∗ Node(n, I_n, I_n) ∗ inFP(n) ∗ inInr(n,k) ∗ k ∉ outs(I_n) }
    n
  | Some n' -> { Inv^N ∗ Node(n, I_n, I_n) ∗ inFP(n) ∗ inInr(n,k) ∗ k ∈ es(I_n, n, n') }
    unlockNode n;
    { Inv^N ∗ inFP(n') ∗ inInr(n',k) }
    traverse n' k
{ v. Inv^N ∗ Node(v, I_v, I_v) ∗ inInr(v,k) ∗ k ∉ outs(I_v) }

let rec searchStrOp ω r k =
  { Inv^N ∗ inFP(r) ∗ inInr(r,k) }
  let n = traverse r k in
  { Inv^N ∗ Node(n, I_n, I_n) ∗ k ∈ ins(I_n, n) ∗ k ∉ outs(I_n) }
  match decisiveOp ω n k with
  | None -> { Inv^N ∗ Node(n, I_n, I_n) ∗ k ∈ ins(I_n, n) ∗ k ∉ outs(I_n) }
    unlockNode n;  { Inv^N }
    searchStrOp ω r k
  | Some res -> { Inv^N ∗ Node(n, I_n, I'_n) ∗ Ψ_ω(k, cn(n), cn'(n), res) }
    { ⌜•I⌝^{γ_I} ⌜cn(I)⌝^{γ_c} ∗ ⌜∘I_n⌝^{γ_I} ∗ ħ(n, I'_n, cn', inr') ∗ Ψ_ω(k, cn(n), cn'(n), res) ⋯ }^{⊤\N}
    (* Linearization point. Execute spec: res' = searchStrSpec ω k *)
    { ⌜•I⌝^{γ_I} ∗ ⌜C'⌝^{γ_c} ∗ ⌜∘I_n⌝^{γ_I} ∗ ħ(n, I'_n, cn', inr') ∗ Ψ_ω(k, cn(n), cn'(n), res) ∗ Ψ_ω(k, cn(dom(I)), C', res') ∗ ⋯ }^{⊤\N}
    { ⌜•I'⌝^{γ_I} ∗ ⌜cn(I')⌝^{γ_c} ∗ ⌜∘I'_n⌝^{γ_I} ∗ ħ(n, I'_n, cn', inr') ∗ res = res' ⋯ }^{⊤\N}
    { Inv^N ∗ ⌜∘I'_n⌝^{γ_I} ∗ ħ(n, I'_n, cn', inr') ∗ res = res' }
    unlockNode n;  { Inv^N ∗ res = res' }
    res
```

Figure 5.5: The link template algorithm with a proof outline.

will be returned by `searchStr0p`. In order to do this, we construct an Iris invariant that relates the implementation state and the specification state and is sufficient to prove correctness:

$$\mathsf{Inv} \coloneqq \exists I.\ \boxed{\bullet I}^{\gamma_I} * \varphi(I) * \boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f} * \mathop{\Large *}_{n \in I} \boxed{\bullet \mathsf{inr}(n)}^{\gamma_{i(n)}} * \boxed{\mathsf{cn}(\mathsf{dom}(I))}^{\gamma_c}$$

$$* \mathop{\Large *}_{n \in I} \exists b.\ \ell(n) \mapsto_i b * (b\ ?\ \mathsf{True} : \exists I_n.\ \mathsf{Node}(n, I_n, I_n))$$

$$\mathsf{Node}(n, I_n, I'_n) \coloneqq \boxed{\circ I_n}^{\gamma_I} * \hbar(n, I'_n, \mathsf{cn}, \mathsf{inr})$$

Our invariant uses a few different types of ghost states in order to capture the state of the link algorithm, the state of the specification, and the relation between them. First, we use the Auth(FI_γ) RA at location γ_I to keep track of the flow graph abstraction. The invariant always owns the authoritative version $\boxed{\bullet I}^{\gamma_I}$, which is the interface of the global graph and that satisfies $\varphi(I)$. We assume that every node $n \in I$ has a lock bit at location $\ell(n)$ that is set to True iff node n is locked. If n is unlocked, the invariant owns the fragment version of n's interface $\boxed{\circ I_n}^{\gamma_I}$ and n's heap representation $\hbar(n, I_n, \mathsf{cn}, \mathsf{inr})$ (denoted $\mathsf{Node}(n, I_n, I_n)$).

We use an authoritative RA of sets of keys, at locations $\gamma_{i(n)}$ for each node n, to encode the inreach of each node. This allows threads to assert that a key is in the inreach of a given node even when it is unlocked, as described in §5.2.2. We also use an authoritative RA of sets of nodes at location γ_f to encode the footprint of the global graph. The invariant owns the authoritative version $\boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f}$, which is the domain of the global interface. This allows threads to take snapshots of the footprint and assert locally that a given node is in the footprint (used for example, in the precondition of `lockNode`).

Finally, we use a sets of keys RA at location γ_c in order to encode the state of the specification program `searchStrSpec`. The relation between the link algorithm and the specification is that the state of the specification must match the set of keys of the implementation, $\mathsf{cn}(\mathsf{dom}(I))$.

In Iris, the invariant assertion $\boxed{\mathsf{Inv}}^{\mathcal{N}}$ denotes state that can be shared between threads that always satisfies the invariant Inv. Invariants are tagged with *name spaces* (our invariant is tagged with the name space $\mathcal{N}$) so as to enforce that the invariant is reestablished in between atomic steps of the program. This is achieved in Iris by tagging the proof goal with a mask $\mathcal{E}$ of available invariants, and allowing the proof to consider an atomic action only when all invariants opened by the previous action have been closed. E.g. the mask $\top \setminus \mathcal{N}$ encodes the fact that $\mathcal{N}$ has been opened and needs to be reestablished before proceeding. We represent this as superscripts on our Hoare-style assertions, and omit the superscript if the mask is $\top$, the set of all invariant names.

Let us now step through the proof of `searchStr0p`. The code begins with a call to `traverse` on line 21. To satisfy `traverse`'s precondition, we need to open the invariant and use the fact that $\varphi(I) \Rightarrow r \in \mathsf{dom}(I)$. We can then take a snapshot of the domain of the global invariant using auth-set-snapshot, and we add $\mathsf{inFP}(r)$ to our context. Note that $\varphi(I) \Rightarrow I^{in}(r)_{\mathsf{is}}(k) = 1$ which by the invariant gives us $k \in \mathsf{inr}(r)$. So we also use $r \in \mathsf{dom}(I)$ to unfold the iterated separating conjunction containing the inreach sets in the invariant, and take a snapshot of the inreach set of r using auth-set-snapshot to add $\mathsf{inInr}(r, k)$ to our context. The resulting context is depicted in line 20.

After the call to `traverse`, we can add its postcondition (line 17) to our context. The next step is the call to `decisiveOp`, for whose precondition we need to show that $k \in \mathsf{ins}(I_n, n)$. This follows from $\mathsf{inInr}(n, k) \land k \notin \mathsf{outs}(I_n)$ by the definition of the invariant, and (5.4).

We then look at the two possible outcomes of `decisiveOp`. In the case where it returns `None`, our context is unchanged, so we execute `unlockNode` using the $\mathsf{Node}(n, I_n, I_n)$ in our context. We can use the specification of `searchStrOp` on the recursive call on line 26 to complete this branch of the proof.

On the other hand, if `decisiveOp` succeeds, we get back a modified node $\mathsf{Node}(n, I_n, I_n')$ with a new interface I_n' that satisfies the search structure specification $\Psi_\omega(k, \mathsf{cn}(n), \mathsf{cn}'(n), \mathsf{res})$ locally (line 27). Since we have modified the search structure, we now have an extra proof obligation for the refinement proof: we need to show that the result value of the specification program if executed at this point is the same as what `searchStrOp` will return. This is essentially the *linearization point* of this algorithm.

To do this, we first open the invariant to get temporary access to its contents, which sets the mask to $\top \setminus \mathcal{N}$ (line 28). We now have the resources to execute the specification program, which changes the specification state from $\mathsf{cn}(n)$ to C' and tells us they satisfy $\Psi_\omega(k, \mathsf{cn}(\mathsf{dom}(I)), C', \mathsf{res}')$.

On the concrete side, we only know $\Psi_\omega(k, \mathsf{cn}(n), \mathsf{cn}'(n), \mathsf{res})$. If we can lift this knowledge to the whole graph, we can use the fact that Ψ_ω determines the result uniquely to obtain $\mathsf{res} = \mathsf{res}'$:

$$\Psi_\omega(k, C, C_1', \mathsf{res}_1) \land \Psi_\omega(k, C, C_2', \mathsf{res}_2) \Rightarrow C_1' = C_2' \land \mathsf{res}_1 = \mathsf{res}_2.$$

Before we do that, we first note that while the node n has interface I_n', the ghost location γ_I still contains the old interface I_n. So we now perform a frame-preserving update to the authoritative flow interface ghost location, using AUTH-FI-UPD and the fact that I_n' contextually extends I_n according to the postcondition of `decisiveOp`. We also need to re-establish the relation between the implementation state and the specification state, namely $\mathsf{cn}(\mathsf{dom}(I')) = C'$. We do this with the help of KEYSET-THM, which gives us $\Psi_\omega(k, \mathsf{cn}(\mathsf{dom}(I)), \mathsf{cn}(\mathsf{dom}(I')), \mathsf{res})$. As Ψ_ω determines the result, we have that $\mathsf{res} = \mathsf{res}'$, completing the refinement proof obligation.

We now have the context in line 31, from which we can close the invariant. We finally execute the call to `unlockNode` as above, and complete the proof. The proof of `traverse` follows a similar line-by-line reasoning using the appropriate specifications of helper functions, except that it does not need to execute the specification program. The intermediate contexts are shown in Figure 5.5.

5.2.4 Lock-coupling and Give-up Templates

In this subsection we list and describe the other templates that we have considered: the give-up and the lock-coupling templates. These correspond to alternate strategies for using locks to avoid unwanted interference between threads. We omit the proofs of these templates because they are very similar to the proof of the link template, use almost identical invariants, and are in fact simpler as neither of these two need the notion of inreach.

Give-up Template The give-up template, like the link template, uses locks only when reading or writing from a node and does not hold locks while traversing from one node to the next.

```
let rec traverse r n k =
  lockNode n;
  if inRange n k then
    match findNext n k with
    | None -> n
    | Some n' -> unlockNode n;
       traverse n' k
  else
    unlockNode n;
    traverse r r k

let rec searchStrOp ω r k =
  let n = traverse r r k in
  match decisiveOp ω n k with
  | None ->
      unlockNode n;
      searchStrOp ω r k
  | Some res ->
      unlockNode n;
      res
```

Figure 5.6: The give-up template algorithm.

```
let rec traverse p n k =
  match findNext n k with
  | None -> (p, n)
  | Some n' ->
      unlockNode p;
      lockNode n;
      traverse n n' k

let rec searchStrOp ω r k =
  lockNode r;
  match findNext r k with
  | None -> searchStrOp ω r k
  | Some n ->
      let (p, n) = traverse r n k in
      let res = decisiveOp ω p n k in
      unlockNode p;
      unlockNode n;
      res
```

Figure 5.7: The lock-coupling template algorithm.

Unlike the link template, there are no link edges added by threads that move data from one node to another. Instead, each node stores a *range* field: this is an under-approximation of that node's inset. Upon arriving at a new node n, each thread locks the node and checks its query key k against the range of n. If k is in the range of n then the thread knows that it is still on the correct path, and it continues. If not, it gives up: it relinquishes the lock on n and goes back to the root of the data structure to retry.

The code for this template algorithm is given in Figure 5.6. Note that apart from the helper functions `findNext` and `decisiveOp`, this template assumes a helper function `inRange`. When called as `inRange n k`, this function returns `true` if and only if k is in the range of n.

The give-up template can be instantiated by a B+ tree, for instance, by adding to each node n additional fields to keep track of lower and upper bounds for keys that are present in the subtree rooted at n. We have also considered a hash table implementation of this template.

Lock-coupling Template The lock-coupling template uses the hand-over-hand locking scheme to ensure that no thread is negatively interfered with while it is on its traversal. Unlike the other two templates, every thread always holds at least one lock while traversing from one node to the next. This means that no other thread can overtake this thread, or perform any modification that would invalidate this thread's search. The code for this template algorithm is given in Fig-

ure 5.7. This is one of the more basic concurrency techniques, and is not as efficient, thus we only consider a linked list implementation of this template.

5.2.5 Proofs of Template Implementations

To obtain a verified implementation of the link template, one needs to provide code for the helper functions in Figure 5.4 that satisfies the given specifications. As can be seen from the specifications, these functions do not have access to the invariant, and hence the shared state, and only have access to the heap representation of the given node. Thus, if their implementations are sequential code, one expects to be able to verify them using an off-the-shelf separation logic tool that can verify sequential heap-manipulating code. Indeed, using some of the core rules of Iris/ReLoC, we can show that we can reason about the implementation program using standard separation logic rules as long as the program only uses those resources that are not locked up in some invariant. Thus, in theory, we can take a proof produced by a standard SL tool and lift it to an Iris proof.

As an example, we present the key definitions that we use to verify the B-link tree implementation in GRASShopper in Figure 5.8. We assume an uninterpreted type of keys `K` (that we axiomatize to be ordered), represent B-link tree nodes using the struct type `Node` and the flow domain and flow interfaces using algebraic data types. The flow domain is axiomatized as described in §5.2.1. Since GRASShopper does not have partial maps, we use the footprint set `dom` to constrain the domains of the inflow and outflow.

`hrepSpatial` is a predicate that defines the access permissions needed for each B-link tree node (it gives permissions to the node and its arrays by means of the access predicate `acc`). The function `edge_fn` is the abstraction function edges that determines the edge function labelling each edge leaving a node. `hrep` is the heap representation predicate $\hbar(n, I_n, \mathsf{cn}, \mathsf{inr})$. For space reasons, we only list the B-link tree specific constraints, such as that the keys are sorted, and omit the parts tying the parameters cn and inr to the heap, and the conditions of γ predicate, as they have been described before.

5.3 Proof Mechanization and Automation

In addition to the link template presented in this thesis, we have also verified the give-up and lock-coupling template algorithms from [Shasha and Goodman 1988], as depicted in Figure 1.2. For the link template and give-up template, we have derived and verified implementations based on B trees and hash tables. For the lock-coupling template we have considered a sorted linked list implementation. The lock-coupling template also captures the synchronization performed by maintenance operations on algorithms such as the split operation on B+ and B-link trees when they traverse the data structure.

The proofs of the template algorithms have been mechanized using the Coq proof assistant, building on the formalization of ReLoC [Frumin et al. 2018]. The implementations of the helper functions for the concrete implementations that are assumed in the template algorithms (e.g. `decisiveOp`, `findNext`, etc.) have been verified using the separation logic based deductive pro-

```
type K

struct Node {
  var len: Int;
  var keys: Array<K>;
  var ptrs: Array<Node>;
  var next: Node;
  ghost var indices: Map<Node, Int>;
}

datatype FlowDom = fd(ks: Map<K, Int>,
  ir: Map<K, Int>)

datatype Interface =
  int(inf: Map<Node, FlowDom>,
      out: Map<Node, FlowDom>,
      dom: Set<Node>)
  | intUndef

predicate hrepSpatial(x: Node) {
  acc(x) * acc(x.keys) * acc(x.ptrs) * x.keys.length == 2*B ∧ x.ptrs.length == 2*B
}

function edge_fn(x: Node, y: Node) returns (e: Map<FlowDom, FlowDom>)
  requires hrepSpatial(x)
  ensures 0 ≤ x.indices[y] ∧ x.indices[y] ≤ x.len
    ⇒ (∀ p: FlowDom, k: K :: e[p].ks[k]
      == ((x.indices[y] == 0 || le(x.keys[x.indices[y] - 1], k))
          ∧ (x.indices[y] == x.len ∧ x.next == null ∧ lt(k, top())
              || lt(k, x.keys[x.indices[y]]))
          ? p.ks[k] : 0 ))
  ensures x.indices[y] == -1
    ⇒ (∀ p: FlowDom, k: K :: e[p].ks[k]
      == (x.next == y ∧ y ≠ null ∧ le(x.keys[x.len], k) ∧ lt(k, top())
          ? p.ks[k] : 0 ))

define hrep(x, I, C, inr) {
  hrepSpatial(x) * x.next ≠ x ∧ I.dom == {x}
    ∧ 0 ≤ x.len < 2*B ∧ (x.next == null ⇒ x.keys[x.len] == top)
    // Outflow determined by edge_fn
    ∧ (∀ z: Node :: fdEq(I.out[z], edge_fn(x, z)[I.inf[x]]))
    // Keys are sorted
    ∧ (∀ i: Int, j: Int :: 0 ≤ i < j ≤ x.len ⇒ x.keys[i] < x.keys[j])
    // All outgoing pointers are distinct
    ∧ (∀ i: Int :: x.ptrs[0] ≠ null ∧ 0 ≤ i ≤ x.len ⇒ x ≠ x.ptrs[i])
    ∧ (∀ i: Int :: x.ptrs[0] ≠ null ∧ 0 ≤ i ≤ x.len ⇒ x.ptrs[i] ≠ x.next)
    ∧ (∀ i: Int, j: Int :: x.ptrs[0] ≠ null ∧ 0 ≤ i < j ≤ x.len ⇒ x.ptrs[i] ≠ x.ptrs[j])
    // Internal nodes don't point to null
    ∧ (∀ i: Int :: x.ptrs[0] ≠ null ∧ 0 ≤ i ≤ x.len ⇒ x.ptrs[i] ≠ null)
    // Indices of outgoing pointers are stored in x.indices
    ∧ (∀ n: Node :: 0 ≤ x.indices[n] ≤ x.len ⇒ x.ptrs.map[x.indices[n]] == n)
    ∧ (∀ i: Int :: x.ptrs[0] ≠ null ∧ 0 ≤ i ≤ x.len ⇒ x.indices[x.ptrs.map[i]] == i)
    ∧ (∀ n: Node :: -1 ≤ x.indices[n] ≤ x.len)
    ∧ (x.ptrs[0] == null ⇒ (∀ n: Node :: x.indices[n] == -1))
    // Tie C and inr to heap, list good conditions:
    ...
}
```

Figure 5.8: B-link tree implementation in GRASShopper.

Table 5.1: Summary of templates and instantiations verified in Iris/Coq and GRASShopper. For each algorithm or library, we show the number of lines of code, lines of proof annotation (including specification), total number of lines, and the proof-checking / verification time in seconds.

Tool	Module	Code	Proof	Total	Checker Time
Iris/Coq	Flow library	-	99	99	-
	Link template	29	518	547	70
	Give-up template	35	406	441	25
	Lock-coupling template	37	565	602	44
	Total	**101**	**1588**	**1689**	**139**
GRASShopper	Flow library	-	127	127	-
	Array library	132	300	432	9
	B+ tree	73	161	234	10
	B-link tree (core)	105	312	417	63
	B-link tree (half split)	56	238	294	167
	B-link tree (full split)	20	186	206	203
	Hash table (link)	58	176	234	7
	Hash table (give-up)	64	141	205	7
	Lock-coupling list	83	231	314	138
	Total	**598**	**1655**	**2253**	**401**

gram verifier GRASShopper [Piskac et al. 2014]. This provided us with a substantial decrease in verification effort, as the tool infers intermediate assertions whose validity is proved automatically using SMT solvers. While we do not have, as of now, a formal proof for the transfer of proofs between Iris and GRASShopper, note that Iris is expressive enough to support all the reasoning that we do in GRASShopper, but comes with significant additional manual effort. Furthermore, our automation of implementation proofs in GRASShopper is another indication that our flow framework is suitable for reasoning in off-the-shelf automated tools.

The Coq formalization assumes that flow interfaces form an RA (Theorem 3.21) and that they enable frame preserving updates (Theorem 3.18) as well as some basic general lemmas about flow interfaces. The template proofs parameterize over the implementation of the helper functions, the heap representation predicate $\hbar$ as well as the actual flow domain, node label domain, and good condition γ.

All properties involving the specific flow domain elements and γ needed in the template proofs are factored out into a few lemmas. These are assumed in Coq and proved in GRASShopper as they can be easily discharged using an SMT solver. To automate the implementation proofs and auxiliary lemmas in GRASShopper, we extended the tool with support for general maps and algebraic data types as they are needed to formalize flow interfaces.

In addition to the helper functions of each data structure that are assumed by the templates we have also verified the split operations for B-link trees. The B-link tree uses a two-part split operation: a half-split that creates a new node, transfers half the contents from a full node to this new node, and adds a link edge; and a full-split that links such newly created nodes to the

tree by adding it to the parent node. For the split operations, we assume a harness template for a maintenance thread that traverses the data structure graph to identify nodes that are amenable to half splits. While we have not verified this harness, we note that it is a simple variation of our lock-coupling template where the abstract specification leaves the contents of the data structure unchanged. For the implementations of half and full splits, we verify that the operation preserves the flow interface of the modified region as well as its contents.

Table 5.1 provides a summary of our development. Experiments have been conducted on an laptop with an Intel Core i7-5600U CPU and 16GB RAM. We split the table into one part for the templates (proved in Coq) and one part for the implementations (proved in GRASShopper). We note that for the B-link tree, B+ tree and hash table implementations, most of the work is done by the array library, which is shared between all these data structures. The size of the proof for the lock-coupling list is relatively large for such a simple data structure. The reason is that the insertion operation, which adds a new node to the list, requires the calculation of a new flow interface for the region obtained after the insertion. This requires the expansion of the definitions of functions related to flow interfaces, which are deeply nested quantified formulas. GRASShopper enforces strict rules that limit quantifier instantiation so as to remain within certain decidable logics [Piskac et al. 2013; Bansal et al. 2015]. Most of the proof in this case involves auxiliary assertions that manually unfold definitions. The actual calculation of the interface is performed by the SMT solver. We note that the size of the proof could be significantly reduced with a few simple tactics for quantifier expansion. Nevertheless, one can see that the additional automation provided by the tool reduces the overall manual proof effort compared to an interactive proof assistant like Coq.

5.4 Related Work

The work presented in this chapter builds on the Iris separation logic [Jung et al. 2018a], the ReLoC logic [Frumin et al. 2018] for expressing refinement proofs in Iris, and the flow framework of compositional abstractions of complex data structures. Our main technical contribution relative to these works is a new proof technique for verifying template algorithms of concurrent search structures that relies on the integration of the flow framework into Iris/ReLoC.

We note that there is no formal connection between the proofs done in Coq and GRASShopper. If one desires end-to-end certified proofs, one can parametrize Iris by the programming language used in GRASShopper and use GRASShopper as an oracle for implementation proofs, or even perform both template and implementation proofs in Iris/Coq (albeit with subtantial manual effort). The only other trusted component is the meta theory of our flow framework. These proofs are simple facts about graphs and graph properties that need only be proven once, apply to every data structure proof in our evaluation and beyond, and are proved in detail in Chapter 3. By contrast, all the proofs pertaining to our individual case studies are performed and verified by either Coq or GRASShopper.

All the algorithms we have considered use fine-grained node-level locking. This is representative of real-world applications, many of which prefer lock-based over lock-free algorithms as

the latter tend to copy data more[2]. On the other hand, our methodology does not require locking, and can be extended to prove lock-free algorithms such as the Bw-tree [Levandoski and Sengupta 2013].

The proofs we obtain are both more modular and simpler than existing proofs of such concurrent data structures. In fact, we are the first to obtain a mechanically verified proof of concurrent B-link trees. Unlike the proof of da Rocha Pinto et al. [2011], which is not mechanized, our proof does not assume node-level operations to be given as primitives. In particular, we also verify the challenging split operation. The only other comparable proof is that of a B+ tree in [Malecha et al. 2010]. However, this work only considers a sequential implementation of B-trees and the proof is considerably more complex than ours (encompassing more than 5000 lines of proof for roughly 500 lines of code).

A recent paper [Meyer and Wolff 2019] demonstrates a similar proof modularity by decoupling the proof of data structures from that of the underlying manual memory management algorithm. Note that as our proofs are done in Iris, which does not support reasoning about deallocation, our proofs assume a garbage collected environment. However, by using Iron [Bizjak et al. 2019], a recent extension of Iris that allows proving absense of memory leaks, we can extend our proofs to the manual memory setting as well. It is a promising direction of future work to integrate this approach and our technique in order to obtain verified data structures where the user can mix-and-match the synchronization technique, memory layout, as well as the memory management system.

There exist many other concurrent separation logics that help modularize the correctness proofs of concurrent systems [Bornat et al. 2005; Heule et al. 2013; Vafeiadis and Parkinson 2007; Feng et al. 2007; Nanevski et al. 2014; Dinsdale-Young et al. 2010; da Rocha Pinto et al. 2014; Xiong et al. 2017; Raad et al. 2015]. Like Iris, their main focus is on modularizing proofs along the interfaces of components of a system (e.g. between the client and implementation of a data structure). Instead, we focus on modularizing the proof of a single component (a concurrent search structure) so that the parts of the proof can be reused across many diverse implementations.

We verify correctness of concurrent data structures by showing that they refine a sequential specification. Most existing techniques instead focus on proving linearizability [Herlihy and Wing 1990]. The connection between contextual refinement and linearizability was established in [Filipovic et al. 2009]. The template algorithms that we have verified focus on lock-based techniques with fixed linearization points inside a decisive operation. The give-up and link templates can be generalized to handle lock-free data structures. Though, many lock-free data structures have non-fixed linearization points, which ReLoC currently cannot reason about. Much work has been dedicated to handling non-fixed as well as external linearization points [O'Hearn et al. 2010; Bouajjani et al. 2017; Chakraborty et al. 2015; Khyzha et al. 2017; Dodds et al. 2015; Liang and Feng 2013; Bouajjani et al. 2013; Zhu et al. 2015; Delbianco et al. 2017]. However, we note that these papers do not aim to separate the proof of thread safety from the proof of structural integrity. In fact, we see our contributions as orthogonal to these works as our approach does

[2]For instance, Apache's CouchDB uses a B+ tree with a global write lock; BerkeleyDB, which has hosted Google's account information, uses a B+ tree with page-level locks in order to trade-off concurrency for better recovery; and java.util.concurrent's hash tables lock the entire list in a bucket during writes, which is more coarse-grained than the one we verify.

not critically depend on the use of ReLoC/Iris. Our proof methodology can be replicated in other separation logics that support user-defined ghost state, such as FCSL [Sergey et al. 2015], which would also be useful if one wanted to extend this work to non-linearizable data structures [Sergey et al. 2016].

Fully automated proofs of linearizability by static analysis and model checking have been mostly confined to simple list-based data structures [Amit et al. 2007; Vafeiadis 2009; Cerný et al. 2010; Dragoi et al. 2013; Abdulla et al. 2013; Bouajjani et al. 2015]. Recent work by Abdulla et al. [2018] shows how to automatically verify more complex structures such as concurrent skip lists that combine lists and arrays. However, it is difficult to devise fully automated techniques that work over a broad class of diverse heap representations. In particular, structures like the B-link tree considered here are still beyond the scope of the state of the art.

5.5 Conclusion

We have presented a proof technique for concurrent search structures that separates the reasoning about thread safety from memory safety. We have demonstrated our technique by formalizing and verifying three template algorithms, and show how to derive verified implementations with significant proof reuse and automation. The result is fully mechanized and partially automated proofs of linearizability and memory safety for a large class of concurrent search structures.

6 Conclusion

This dissertation set out to bridge the gap between the concurrent data structures used in the real world and the algortihms that can be verified by formal reasoning techniques. Towards this goal, the disseration introduced the flow framework, a novel approach to the abstraction and specification of unbounded data structures in a way that permits modular reasoning. The framework simplifies existing proofs, brings the proofs of complex concurrent data structures within reach to automated and mechanized verification tools, and opens up the possibility of template-based verification to further modularize proofs of certain important classes of concurrent data structures. Flows can be used within existing separation logic systems and tools, and also permit a large degree of automation. These contributions make significant strides towards the practical verification of real-world concurrent algorithms.

6.1 Future Work

The ideas and contributions in this dissertation suggest a number of interesting avenues for future work.

Flow Framework In terms of the theory of flows, one interesting question is to investigate the potential for an extended meta-theory supporting syncing of partial dirty regions, as an alternative proof technique to our edge-local transformation. It is also desirable to connect our techniques to other verification tools, and to mechanise the theoretical foundations.

Automation An obvious first-step in increasining the level of automation of flow-based proofs is to develop a lightweight front-end tool, facilitating the application of flow-based reasoning to further and more complex examples. Building from here, the next natural question is the inference of flow-based specifications. We have shown how a small number of flow domains can be combined to represent a variety of common data structures. Building on the large body of work on inferring invariants for heap-manipulating programs (e.g., [Sagiv et al. 2002]), including techniques based on separation logic [Distefano et al. 2006; Calcagno et al. 2009; Vafeiadis 2010], it would be interesting to see if techniques from such shape analyses (typically using some form of abstract interpretation [Cousot and Cousot 1977]) can be extended to infer flow-based specifications.

Templates for Other ADTs The template-based modularity presented in Chapter 5 was limited to search structures. While this is a large and important class of concurrent data structures, an interesting question is to see if one can come up with template algorithms and proofs for other ADTs, such as queues and stacks.

Manual Memory Management A recent paper [Meyer and Wolff 2019] demonstrates a similar proof modularity by decoupling the proof of data structures from that of the underlying manual memory management algorithm. Note that as our proofs are done in Iris, which does not support reasoning about deallocation, our proofs assume a garbage collected environment. However, by using Iron [Bizjak et al. 2019], a recent extension of Iris that allows proving absense of memory leaks, we can extend our proofs to the manual memory setting as well. It is a promising direction of future work to integrate this approach and our technique in order to obtain verified data structures where the user can mix-and-match the synchronization technique, memory layout, as well as the memory management system.

A Appendix

A.1 Encoding of Flows in Viper

We list here the encoding of the flow framework in Viper.

```
// Interfaces
domain Interface {
  // Constructor
  function interface(i: Map[Ref, FlowDom], o: Map[Ref, FlowDom],
    F: Set[Ref]) : Interface
  // Destructors
  function inf(I: Interface) : Map[Ref, FlowDom] // inflow
  function out(I: Interface) : Map[Ref, FlowDom] // outflow
  function dom(I: Interface) : Set[Ref] // footprint

  // Validity
  function intValid(I: Interface) : Bool

  axiom interfaceDef {
    forall i: Map[Ref, FlowDom], o: Map[Ref, FlowDom],
        F: Set[Ref] :: {interface(i, o, F)}
      inf(interface(i, o, F)) == i
      && out(interface(i, o, F)) == o
      && dom(interface(i, o, F)) == F
  }

  axiom injective_interface {
    forall I: Interface :: {inf(I), out(I), dom(I)}
      I == interface(inf(I), out(I), dom(I))
  }

  // Interface contextual extension (allows increasing domain)
  function intLeq(I1: Interface, I2: Interface) : Bool

  axiom intLeq_eq {
    forall I: Interface :: {intLeq(I, I)} intLeq(I, I)
  }

  axiom intLeq_implies {
```

```
35      forall I1: Interface, I2: Interface :: {intLeq(I1, I2)}
36        intLeq(I1, I2)
37        ==> intValid(I1) && intValid(I2)
38          && dom(I1) subset dom(I2)
39          && (forall n: Ref :: {Iinf(I2, n)}
40            n in dom(I1) ==> fdEq(Iinf(I1, n), Iinf(I2, n)))
41          && (forall m: Ref :: {Iout(I2, m)}
42            !(m in dom(I2)) ==> fdEq(Iout(I1, m), Iout(I2, m)))
43    }
44
45    axiom intLeq_transitive {
46      forall I1: Interface, I2: Interface, I3: Interface ::
47          {intLeq(I1, I2), intLeq(I2, I3), intLeq(I1, I3)}
48        intLeq(I1, I2) && intLeq(I2, I3) ==> intLeq(I1, I3)
49    }
50  }
51
52  // Macros to simplify specs
53  define Iinf(I, x) select(inf(I), x)
54  define Iout(I, y) select(out(I), y)
55
56  // Helper lemma to prove validity of singleton interfaces
57  method _lemma_prove_intValid(yy: Ref)
58    requires nodeSpatial(yy) && acc(yy.intf) && dom(yy.intf) == Set(yy)
59    // Outflow consistent with edge_fn
60    requires (forall z: Ref :: {Iout(yy.intf, z)}{edge_fn(yy, z)}
61      fdEq(Iout(yy.intf, z), select(edge_fn(yy, z),Iinf(yy.intf, yy))))
62    ensures nodeSpatial(yy) && acc(yy.intf)
63    ensures hrepUnchanged(yy) && yy.intf == old(yy.intf)
64    ensures intValid(intCompSet(Set(yy)))
65
66  // Helper lemma to prove intLeq
67  method _lemma_prove_intLeq(I1: Interface, I2: Interface)
68    requires dom(I1) subset dom(I2)
69    requires intValid(I1) && intValid(I2)
70    requires (forall n: Ref :: {Iinf(I2, n)}
71          n in dom(I1) ==> fdEq(Iinf(I1, n), Iinf(I2, n)))
72    requires (forall m: Ref :: {Iout(I2, m)}
73          !(m in dom(I2)) ==> fdEq(Iout(I1, m), Iout(I2, m)))
74    ensures intLeq(I1, I2)
75
76  // Apply the Replacement Theorem
77  define _replacement_theorem(X, X1, Y, Y1, pre) {
78    // This is an instantiation of the Replacement Theorem:
79    assume intLeq(old[pre](intCompSet(Y)), intCompSet(Y1))
80      ==> intLeq(old[pre](intCompSet(X)), intCompSet(X1))
81    // By defn of composition, new nodes' inflow given by inflow of method's footprint
82    assume forall z: Ref :: {Iinf(intCompSet(X1), z)}
83      z in (X1 setminus X) ==> Iinf(intCompSet(X1), z) == Iinf(intCompSet(Y1), z)
84  }
85
```

```
// ---- Basic elements of encoding: intf field and N predicate

field intf: Interface

// This is defined outside the domain as it is state-dependent
function intCompSet(X: Set[Ref]) : Interface
  requires forall xx: Ref :: {xx.intf} xx in X ==> acc(xx.intf)
  ensures dom(result) == X
  // The empty interface
  ensures X == Set[Ref]()
    ==> (forall xx: Ref :: {Iinf(result, xx)} fdEq(Iinf(result, xx), fdZero()))
  ensures X == Set[Ref]()
    ==> (forall xx: Ref :: {Iout(result, xx)} fdEq(Iout(result, xx), fdZero()))
  ensures X == Set[Ref]() ==> intValid(result)
  // The singleton interface
  ensures forall xx: Ref :: {intCompSet(Set(xx))} X == Set(xx) && xx in X ==> result == xx.intf

define node(x)
  nodeSpatial(x) && acc(x.intf) && dom(x.intf) == Set(x)
  && (forall z: Ref :: {Iout(x.intf, z)}{edge_fn(x, z)}
    fdEq(Iout(x.intf, z), select(edge_fn(x, z),Iinf(x.intf, x))))


// ---- Tricks to compute unbounded sums

// Compute capacity (approximately)
domain Cap {
  function mapPlus(m1: Map[FlowDom, FlowDom], m2: Map[FlowDom, FlowDom])
    : Map[FlowDom, FlowDom]
  axiom mapPlusLookup {
    forall m1: Map[FlowDom, FlowDom], m2: Map[FlowDom, FlowDom], f: FlowDom ::
        {select(mapPlus(m1, m2), f)}
      select(mapPlus(m1, m2), f) == fdPlus(select(m1, f), select(m2, f))
  }

  function capAux(m: Ref, n: Ref, edges: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]],
    through: List[Ref]) : Map[FlowDom, FlowDom]

  axiom capDirect {
    forall m: Ref, n: Ref, edges: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]] ::
        {capAux(m, n, edges, nil())}
      capAux(m, n, edges, nil()) == select(select(edges, m), n)
  }

  axiom capMaybeThrough {
    forall m: Ref, n: Ref, edges: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]],
        i: Ref, rest: List[Ref] ::
        {capAux(m, n, edges, cons(i, rest))}
      capAux(m, n, edges, cons(i, rest))
      == mapPlus(capAux(m, n, edges, rest),
          pipe(capAux(m, i, edges, rest), capAux(i, n, edges, rest)))
```

```
  }
}
define cap(n0, n1, m, d) capAux(n0, n1, m, remove(remove(d, n0), n1))

// The sum of outflow from a given region
domain OutflowSum {
  function outflowAux(n: Ref, flows: Map[Ref, FlowDom],
    edges: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]], l: List[Ref]) : FlowDom

  axiom OutflowSumBase {
    forall n: Ref, flows: Map[Ref, FlowDom],
        edges: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]] ::
        {outflowAux(n, flows, edges, nil())}
      outflowAux(n, flows, edges, nil()) == fdZero()
  }

  axiom OutflowSumMaybeThrough {
    forall n: Ref, flows: Map[Ref, FlowDom],
        edges: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]], m: Ref, rest: List[Ref] ::
        {outflowAux(n, flows, edges, cons(m, rest))}
      outflowAux(n, flows, edges, cons(m, rest))
        == fdPlus(outflowAux(n, flows, edges, rest),
           select(select(select(edges, m), n), select(flows, m))) // flow(m) |> edge(m, n)
  }
}
define outflowSum(n, f, e, D) outflowAux(n, f, e, remove(D, n))


// ---- Macros for flow annotations to be automatically added:

// (Re)-initialize all variables
define _init_vars() {
  dirty := Set[Ref]()
  dirtyList := nil()
  // edge_fns[n][n'] is e(n, n')
  edge_fns := havoc_map_map_edge()
  // edge_fns_new[n][n'] is e'(n, n')
  edge_fns_new := havoc_map_map_edge()
  // infs_small[n] is In.inf[n]
  infs_small := havoc_map()
  // infs_small_new[n] is In'.inf[n]
  infs_small_new := havoc_map()
  // inf_D is inflow of interface of dirty region
  inf_D := havoc_map()
}

// Add n to the dirty region
define _make_dirty(n0) { // could also parameterise by assigned-to variables
  if(n0 != null && !(n0 in dirty)) {
    dirty := dirty union Set(n0)
    dirtyList := cons(n0, dirtyList)
```

```
188   }
189 }
190
191 define _set_edge_fns(edge_fns, X) {
192   assume forall x: Ref, y: Ref :: {select(select(edge_fns, x), y)}
193     x in X ==> select(select(edge_fns, x), y) == edge_fn(x, y)
194 }
195
196 define _set_infs_and_fms(infs_small, edge_fns_new, X) {
197   assume forall x: Ref :: {select(infs_small, x)}
198     x in dirty ==> select(infs_small, x) == Iinf(x.intf, x)
199   _set_edge_fns(edge_fns_new, X)
200 }
201
202 define _assume_flow_eqn(infs_small, edge_fns, inf_D) {
203   assume forall x: Ref :: {select(infs_small, x)} {select(inf_D, x)}
204     x in dirty ==> select(infs_small, x)
205       == fdPlus(select(inf_D, x), outflowSum(x, infs_small, edge_fns, dirtyList))
206 }
207
208 // General flow framework lemmas
209 define _assume_flow_lemmas(X) {
210   assume forall x: Ref :: {Iinf(x.intf, x)}
211     x in X && intValid(intCompSet(X))
212     ==> fdLeq(Iinf(intCompSet(X), x), Iinf(x.intf, x))
213   assume forall x: Ref, y: Ref :: {trig_gfe(x, y)}
214     x in X && y in X && intValid(intCompSet(X))
215     ==> fdLeq(Iout(x.intf, y), Iinf(y.intf, y))
216 }
217
218 // Trigger for global flow equation
219 function trig_gfe(x: Ref, y: Ref) : Bool { true }
220
221 define _field_read(x, xf) {
222   assert trig_gfe(x, xf)
223 }
224
225 // Assume acyclic based on infs_small (i.e. actual flow), because
226 // this is what is known about the pre state
227 define _assume_no_cycles(infs_small, edge_fns) {
228   assume (forall x: Ref :: {select(infs_small, x)} x in dirty
229     ==> select(cap(x, x, edge_fns, dirtyList), select(infs_small, x))
230         == fdZero())
231 }
232
233 // Assert acyclic based on inf_D because this must be the same in the post,
234 // and infs_small_new only determined by flow eqn (which is only true if acyclic)
235 define _assert_no_cycles(inf_D, edge_fns_new) {
236   assert (forall x: Ref, y: Ref :: {no_match(x), no_match(y)}
237     x in dirty && y in dirty
238     ==> fdEq(
```

```
  select(pipe(cap(x, y, edge_fns_new, dirtyList),
              cap(y, y, edge_fns_new, dirtyList)),
         select(inf_D, x)),
             fdZero()))
}

// Check path subflow preservation for dirty region
// This is needed to guarantee that composite graph is also EA
define _assert_path_subflow_preserved(inf_D, edge_fns, edge_fns_new) {
  assert forall x: Ref, z: Ref, p: FlowDom ::
      {no_match(x), no_match(z), no_match_fd(p)}
    x in dirty && !(z in dirty) && fdLeq(p, select(inf_D, x))
    ==> fdEq(select(cap(x, z, edge_fns, dirtyList), p),
          select(cap(x, z, edge_fns_new, dirtyList), p))
}

define _pre_code(FP) {
  _init_vars()
  _assume_flow_lemmas(FP)
  _set_edge_fns(edge_fns, FP)
}

// Compute footprints after a method call
define _compute_footprints(X, X1, Y, Y1) {
  X1 := X union (Y1 setminus Y)
}

define _post_method_call(FP, FP1, Y, Y1, pre) {
  _compute_footprints(FP, FP1, Y, Y1)
  _replacement_theorem(FP, FP1, Y, Y1, pre)
}

define _post_code_sync(FP) {
  // Summarise infs and fms of singleton interfaces in dirty region
  _set_infs_and_fms(infs_small, edge_fns_new, FP)

  // Assume flow equation on pre state
  _assume_flow_eqn(infs_small, edge_fns, inf_D)

  // Assume flow equation on new state
  _assume_flow_eqn(infs_small_new, edge_fns_new, inf_D)

  // Check equivalence of flow maps
  label pre_sync
  // Check node for each node in dirty and write new interface to x.intf
  sync(dirty, dirtyList, inf_D, infs_small, infs_small_new, edge_fns, edge_fns_new)
  _replacement_theorem(FP, FP, dirty, dirty, pre_sync)
}

define _post_code_sync_EA(FP) {
  // Summarise infs and fms of singleton interfaces in dirty region
```

```
  _set_infs_and_fms(infs_small, edge_fns_new, FP)

  // Assume flow equation on pre state
  _assume_flow_eqn(infs_small, edge_fns, inf_D)

  // Assume that dirty region had no cycles
  _assume_no_cycles(infs_small, edge_fns)

  // Check that dirty region has no cycles
  _assert_no_cycles(inf_D, edge_fns_new)

  // Assume flow equation on new state
  _assume_flow_eqn(infs_small_new, edge_fns_new, inf_D)

  // Check that we won't introduce any cycles in the larger graph
  _assert_path_subflow_preserved(inf_D, edge_fns, edge_fns_new)

  // Check equivalence of flow maps
  label pre_sync
  // Check node for each node in dirty and write new interface to x.intf
  sync(dirty, dirtyList, inf_D, infs_small, infs_small_new, edge_fns, edge_fns_new)
  _replacement_theorem(FP, FP, dirty, dirty, pre_sync)
}


// ---- Flow-specific ghost procedures:

// Sync the dirty set of nodes and update their intf fields
// Requires the interface of dirty to be exactly equal
method sync(dirty: Set[Ref], dirtyList: List[Ref],
            inf_D: Map[Ref, FlowDom],
            infs_small: Map[Ref, FlowDom], infs_small_new: Map[Ref, FlowDom],
            edge_fns: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]],
            edge_fns_new: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]])

  requires forall x1: Ref :: {no_match(x1)} x1 in dirty ==> acc(x1.intf)
  // Check that new interface of dirty == old interface of dirty
  // We know that dom and inf is equal by construction, so check outflow preserved:
  requires forall z: Ref :: {no_match(z)} !(z in dirty)
    ==> fdEq(outflowSum(z, infs_small, edge_fns, dirtyList),
      outflowSum(z, infs_small_new, edge_fns_new, dirtyList))

  ensures forall x1: Ref :: {x1 in dirty}
    x1 in dirty ==> acc(x1.intf) && dom(x1.intf) == Set(x1)
  // Now x.intf contains the interface given by infs_small_new and fms_small_new
  ensures forall x1: Ref :: {Iinf(x1.intf, x1)}
    x1 in dirty ==> Iinf(x1.intf, x1) == select(infs_small_new, x1)
  ensures forall x1: Ref, y1: Ref :: {Iout(x1.intf, y1)}
    x1 in dirty ==> Iout(x1.intf, y1)
      == select(select(select(edge_fns_new, x1), y1), Iinf(x1.intf, x1))
  // and the interface is the same
```

```
341   ensures old(intCompSet(dirty)) == intCompSet(dirty)
342
343
344 // ---- Misc util stuff:
345
346 // Nondeterministic boolean value
347 method nondet() returns (b: Bool)
348
349 // Havoc a flow map
350 method havoc_map() returns (m: Map[Ref, FlowDom])
351 method havoc_map_map() returns (m: Map[Ref, Map[Ref, FlowDom]])
352 method havoc_map_map_edge() returns (m: Map[Ref, Map[Ref, Map[FlowDom, FlowDom]]])
353
354 // Trigger that's never triggered
355 domain NoMatching {
356   function no_match(r: Ref) : Bool
357   function no_match_int(x: Int) : Bool
358   function no_match_fd(d: FlowDom) : Bool
359 }
360
361 // Dummy term to seed triggers
362 function dummy(f: FlowDom) : Bool { true }
```

Bibliography

Abdulla, P. A., Haziza, F., Holík, L., Jonsson, B., and Rezine, A. (2013). An integrated specification and verification technique for highly concurrent data structures. In Piterman, N. and Smolka, S. A., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 324–338. Springer.

Abdulla, P. A., Jonsson, B., and Trinh, C. Q. (2018). Fragment abstraction for concurrent shape analysis. In Ahmed, A., editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 442–471. Springer.

Amit, D., Rinetzky, N., Reps, T. W., Sagiv, M., and Yahav, E. (2007). Comparison under abstraction for verifying linearizability. In Damm, W. and Hermanns, H., editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer.

Appel, A. W. (2012). Verified software toolchain. In Goodloe, A. and Person, S., editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, page 2. Springer.

Bansal, K., Reynolds, A., King, T., Barrett, C. W., and Wies, T. (2015). Deciding local theory extensions via e-matching. In Kroening, D. and Pasareanu, C. S., editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 87–105. Springer.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2004). A decidable fragment of separation logic. In Lodaya, K. and Mahajan, M., editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer.

Birkedal, L. and Bizjak, A. (2018). Lecture notes on iris: Higher-order concurrent separation logic.

Bizjak, A., Gratzer, D., Krebbers, R., and Birkedal, L. (2019). Iron: managing obligations in higher-order concurrent separation logic. *PACMPL*, 3(POPL):65:1–65:30.

Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. (2005). Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA. ACM.

Bornat, R., Calcagno, C., and Yang, H. (2006). Variables as resource in separation logic. *Electron. Notes Theor. Comput. Sci.*, 155:247–276.

Bouajjani, A., Dragoi, C., Enea, C., and Sighireanu, M. (2012). Accurate invariant checking for programs manipulating lists and arrays with infinite data. In Chakraborty, S. and Mukund, M., editors, *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 167–182. Springer.

Bouajjani, A., Emmi, M., Enea, C., and Hamza, J. (2013). Verifying concurrent programs against sequential specifications. In Felleisen, M. and Gardner, P., editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 290–309. Springer.

Bouajjani, A., Emmi, M., Enea, C., and Hamza, J. (2015). On reducing linearizability to state reachability. In Halldórsson, M. M., Iwama, K., Kobayashi, N., and Speckmann, B., editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 95–107. Springer.

Bouajjani, A., Emmi, M., Enea, C., and Mutluergil, S. O. (2017). Proving linearizability using forward simulations. In Majumdar, R. and Kuncak, V., editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer.

Brookes, S. (2007). A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270.

Brookes, S. and O'Hearn, P. W. (2016). Concurrent separation logic. *SIGLOG News*, 3(3):47–65.

Brotherston, J., Distefano, D., and Petersen, R. L. (2011). Automated cyclic entailment proofs in separation logic. In Bjørner, N. and Sofronie-Stokkermans, V., editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 131–146. Springer.

Burckhardt, S., Alur, R., and Martin, M. M. K. (2007). Checkfence: checking consistency of concurrent data types on relaxed memory models. In Ferrante, J. and McKinley, K. S., editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 12–21. ACM.

Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P. W., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. (2015). Moving fast with software verification. In Havelund, K., Holzmann, G. J., and Joshi, R., editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer.

Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. (2009). Compositional shape analysis by means of bi-abduction. In Shao, Z. and Pierce, B. C., editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM.

Calcagno, C., O'Hearn, P. W., and Yang, H. (2007). Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 366–378. IEEE Computer Society.

Cao, Q., Cuellar, S., and Appel, A. W. (2017). Bringing order to the separation logic jungle. In Chang, B. E., editor, *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, volume 10695 of *Lecture Notes in Computer Science*, pages 190–211. Springer.

Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., and Alur, R. (2010). Model checking of linearizability of concurrent list implementations. In Touili, T., Cook, B., and Jackson, P. B., editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 465–479. Springer.

Chakraborty, S., Henzinger, T. A., Sezgin, A., and Vafeiadis, V. (2015). Aspect-oriented linearizability proofs. *Logical Methods in Computer Science*, 11(1).

Chlipala, A. (2011). Mostly-automated verification of low-level programs in computational separation logic. In Hall, M. W. and Padua, D. A., editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 234–245. ACM.

Cook, B., Haase, C., Ouaknine, J., Parkinson, M. J., and Worrell, J. (2011). Tractable reasoning in a fragment of separation logic. In Katoen, J. and König, B., editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 235–249. Springer.

Coq Development Team, T. (2017). *The Coq Proof Assistant Reference Manual, version 8.7.*

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*.

da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., and Wheelhouse, M. J. (2011). A simple abstraction for complex concurrent indexes. In Lopes, C. V. and Fisher, K., editors,

Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, pages 845–864. ACM.

da Rocha Pinto, P., Dinsdale-Young, T., and Gardner, P. (2014). Tada: A logic for time and data abstraction. In Jones, R. E., editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer.

Delbianco, G. A., Sergey, I., Nanevski, A., and Banerjee, A. (2017). Concurrent data structures linked in time. In Müller, P., editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 8:1–8:30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Detlefs, D., Nelson, G., and Saxe, J. B. (2005). Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473.

Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J., and Yang, H. (2013). Views: compositional reasoning for concurrent programs. In Giacobazzi, R. and Cousot, R., editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 287–300. ACM.

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. (2010). Concurrent abstract predicates. In D'Hondt, T., editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer.

Distefano, D., O'Hearn, P. W., and Yang, H. (2006). A local shape analysis based on separation logic. In Hermanns, H. and Palsberg, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer.

Dockins, R., Hobor, A., and Appel, A. W. (2009). A fresh look at separation algebras and share accounting. In Hu, Z., editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 161–177. Springer.

Dodds, M., Haas, A., and Kirsch, C. M. (2015). A scalable, correct time-stamped stack. In Rajamani, S. K. and Walker, D., editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 233–246. ACM.

Dodds, M., Jagannathan, S., Parkinson, M. J., Svendsen, K., and Birkedal, L. (2016). Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2):4:1–4:72.

Dragoi, C., Gupta, A., and Henzinger, T. A. (2013). Automatic linearizability proofs of concurrent objects with cooperating updates. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 174–190. Springer.

Enea, C., Lengál, O., Sighireanu, M., and Vojnar, T. (2017). SPEN: A solver for separation logic. In Barrett, C. W., Davies, M., and Kahsai, T., editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 302–309.

Enea, C., Sighireanu, M., and Wu, Z. (2015). On automated lemma generation for separation logic with inductive definitions. In Finkbeiner, B., Pu, G., and Zhang, L., editors, *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, volume 9364 of *Lecture Notes in Computer Science*, pages 80–96. Springer.

Feng, X., Ferreira, R., and Shao, Z. (2007). On the relationship between concurrent separation logic and assume-guarantee reasoning. In Nicola, R. D., editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer.

Filipovic, I., O'Hearn, P. W., Rinetzky, N., and Yang, H. (2009). Abstraction for concurrent objects. In Castagna, G., editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 252–266. Springer.

Frumin, D., Krebbers, R., and Birkedal, L. (2018). Reloc: A mechanised relational logic for fine-grained concurrency. In Dawar, A. and Grädel, E., editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 442–451. ACM.

Fu, M., Li, Y., Feng, X., Shao, Z., and Zhang, Y. (2010). Reasoning about optimistic concurrency using a program logic for history. In Gastin, P. and Laroussinie, F., editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 388–402. Springer.

Harris, T. L. (2001). A pragmatic implementation of non-blocking linked-lists. In Welch, J. L., editor, *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer.

Herlihy, M. and Shavit, N. (2008). *The art of multiprocessor programming*. Morgan Kaufmann.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

Heule, S., Leino, K. R. M., Müller, P., and Summers, A. J. (2013). Abstract read permissions: Fractional permissions without the fractions. In Giacobazzi, R., Berdine, J., and Mastroeni, I., editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 315–334. Springer.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Hobor, A. and Villard, J. (2013). The ramifications of sharing in data structures. In Giacobazzi, R. and Cousot, R., editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 523–536. ACM.

Immerman, N., Rabinovich, A. M., Reps, T. W., Sagiv, S., and Yorsh, G. (2004). The boundary between decidability and undecidability for transitive-closure logics. In Marcinkowski, J. and Tarlecki, A., editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer.

Iosif, R., Rogalewicz, A., and Vojnar, T. (2014). Deciding entailments in inductive separation logic with tree automata. In Cassez, F. and Raskin, J., editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 201–218. Springer.

Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., and Sagiv, M. (2013). Effectively-propositional reasoning about reachability in linked data structures. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 756–772. Springer.

Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. (2011). Verifast: A powerful, sound, predictable, fast verifier for C and java. In Bobaru, M. G., Havelund, K., Holzmann, G. J., and Joshi, R., editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer.

Jones, C. B. (1983). Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332.

Jung, R., Krebbers, R., Birkedal, L., and Dreyer, D. (2016). Higher-order ghost state. In Garrigue, J., Keller, G., and Sumii, E., editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM.

Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. (2018a). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20.

Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. (2018b). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20.

Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., and Dreyer, D. (2015). Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Rajamani, S. K. and Walker, D., editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM.

Katelaan, J., Matheja, C., and Zuleger, F. (2019). Effective entailment checking for separation logic with inductive definitions. In Vojnar, T. and Zhang, L., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, volume 11428 of *Lecture Notes in Computer Science*, pages 319–336. Springer.

Khyzha, A., Dodds, M., Gotsman, A., and Parkinson, M. J. (2017). Proving linearizability using partial orders. In Yang, H., editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 639–667. Springer.

Klarlund, N. and Schwartzbach, M. I. (1993). Graph types. In Deusen, M. S. V. and Lang, B., editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 196–205. ACM Press.

Krebbers, R., Jung, R., Bizjak, A., Jourdan, J., Dreyer, D., and Birkedal, L. (2017). The essence of higher-order concurrent separation logic. In Yang, H., editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer.

Krishna, S., Shasha, D. E., and Wies, T. (2018a). Go with the flow: Compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.*, 2(POPL):37:1–37:31.

Krishna, S., Shasha, D. E., and Wies, T. (2018b). Go with the flow: compositional abstractions for concurrent data structures. *PACMPL*, 2(POPL):37:1–37:31.

Lahiri, S. K. and Qadeer, S. (2008). Back to the future: revisiting precise program verification using SMT solvers. In Necula, G. C. and Wadler, P., editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 171–182. ACM.

Lehman, P. L. and Yao, S. B. (1981). Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670.

Levandoski, J. J. and Sengupta, S. (2013). The bw-tree: A latch-free b-tree for log-structured flash storage. *IEEE Data Eng. Bull.*, 36(2):56–62.

Liang, H. and Feng, X. (2013). Modular verification of linearizability with non-fixed linearization points. In Boehm, H. and Flanagan, C., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 459–470. ACM.

Madhusudan, P., Qiu, X., and Stefanescu, A. (2012). Recursive proofs for inductive tree data-structures. In Field, J. and Hicks, M., editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 123–136. ACM.

Malecha, J. G., Morrisett, G., Shinnar, A., and Wisnesky, R. (2010). Toward a verified relational database management system. In Hermenegildo, M. V. and Palsberg, J., editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 237–248. ACM.

Meyer, R. and Wolff, S. (2019). Decoupling lock-free data structures from memory reclamation for static analysis. *PACMPL*, 3(POPL):58:1–58:31.

Michael, M. and Scott, M. (1995). Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester.

Müller, P., Schwerhoff, M., and Summers, A. J. (2016). Automatic verification of iterated separating conjunctions using symbolic execution. In Chaudhuri, S. and Farzan, A., editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 405–425. Springer.

Müller, P., Schwerhoff, M., and Summers, A. J. (2016). Viper: A verification infrastructure for permission-based reasoning. In Jobstmann, B. and Leino, K. R. M., editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag.

Nanevski, A., Ley-Wild, R., Sergey, I., and Delbianco, G. A. (2014). Communicating state transition systems for fine-grained concurrent resources. In Shao, Z., editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer.

Nguyen, H. H. and Chin, W. (2008). Enhancing program verification with lemmas. In Gupta, A. and Malik, S., editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 355–369. Springer.

O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307.

O'Hearn, P. W. and Pym, D. J. (1999). The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. (2001). Local reasoning about programs that alter data structures. In Fribourg, L., editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer.

O'Hearn, P. W., Rinetzky, N., Vechev, M. T., Yahav, E., and Yorsh, G. (2010). Verifying linearizability with hindsight. In Richa, A. W. and Guerraoui, R., editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 85–94. ACM.

Owicki, S. S. and Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285.

Parkinson, M. (2007). Class invariants: The end of the road? In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*.

Parkinson, M. J. and Summers, A. J. (2012). The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3).

Pek, E., Qiu, X., and Madhusudan, P. (2014). Natural proofs for data structure manipulation in C using separation logic. In O'Boyle, M. F. P. and Pingali, K., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 440–451. ACM.

Pérez, J. A. N. and Rybalchenko, A. (2011). Separation logic + superposition calculus = heap theorem prover. In Hall, M. W. and Padua, D. A., editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 556–566. ACM.

Piskac, R., Wies, T., and Zufferey, D. (2013). Automating separation logic using SMT. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789. Springer.

Piskac, R., Wies, T., and Zufferey, D. (2014). Grasshopper - complete heap verification with mixed specifications. In Ábrahám, E. and Havelund, K., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 124–139. Springer.

Qiu, X. and Wang, Y. (2019). A decidable logic for tree data-structures with measurements. In Enea, C. and Piskac, R., editors, *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, volume 11388 of *Lecture Notes in Computer Science*, pages 318–341. Springer.

Raad, A., Hobor, A., Villard, J., and Gardner, P. (2016). Verifying concurrent graph algorithms. In Igarashi, A., editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 314–334.

Raad, A., Villard, J., and Gardner, P. (2015). Colosl: Concurrent local subjective logic. In Vitek, J., editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 710–735. Springer.

Reynolds, A., Iosif, R., and Serban, C. (2017). Reasoning in the bernays-schönfinkel-ramsey fragment of separation logic. In Bouajjani, A. and Monniaux, D., editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 462–482. Springer.

Reynolds, J. C. (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*, pages 55–74. IEEE Computer Society.

Sagiv, S., Reps, T. W., and Wilhelm, R. (2002). Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298.

Sergey, I., Nanevski, A., and Banerjee, A. (2015). Mechanized verification of fine-grained concurrent programs. In Grove, D. and Blackburn, S., editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 77–87. ACM.

Sergey, I., Nanevski, A., Banerjee, A., and Delbianco, G. A. (2016). Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In Visser, E. and Smaragdakis, Y., editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 92–110. ACM.

Sha, L., Rajkumar, R., and Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185.

Shasha, D. E. and Goodman, N. (1988). Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90.

Smans, J., Jacobs, B., and Piessens, F. (2009). Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009*, volume 5653 of *LNCS*, pages 148–172.

Summers, A. J. and Drossopoulou, S. (2010). Considerate reasoning and the composite design pattern. In Barthe, G. and Hermenegildo, M. V., editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 328–344. Springer.

Svendsen, K. and Birkedal, L. (2014). Impredicative concurrent abstract predicates. In Shao, Z., editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer.

Tatsuta, M., Le, Q. L., and Chin, W. (2016). Decision procedure for separation logic with inductive definitions and presburger arithmetic. In Igarashi, A., editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 423–443.

Ter-Gabrielyan, A., Müller, P., and Summers, A. J. (2020). Modular verification of heap reachability properties in separation logic. In *Proceedings of OOPSLA 2020*. Conditionally Accepted.

Vafeiadis, V. (2009). Shape-value abstraction for verifying linearizability. In Jones, N. D. and Müller-Olm, M., editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer.

Vafeiadis, V. (2010). Rgsep action inference. In Barthe, G. and Hermenegildo, M. V., editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 345–361. Springer.

Vafeiadis, V. and Parkinson, M. J. (2007). A marriage of rely/guarantee and separation logic. In Caires, L. and Vasconcelos, V. T., editors, *CONCUR 2007 - Concurrency Theory, 18th International*

Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer.

Wies, T., Muñiz, M., and Kuncak, V. (2011). An efficient decision procedure for imperative tree data structures. In Bjørner, N. and Sofronie-Stokkermans, V., editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 476–491. Springer.

Xiong, S., da Rocha Pinto, P., Ntzik, G., and Gardner, P. (2017). Abstract specifications for concurrent maps. In Yang, H., editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 964–990. Springer.

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

Yang, H. (2001b). *Local reasoning for stateful programs*. University of Illinois at Urbana-Champaign.

Zhu, H., Petri, G., and Jagannathan, S. (2015). Poling: SMT aided linearizability proofs. In Kroening, D. and Pasareanu, C. S., editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer.