Make flows small again: revisiting the flow framework

Roland Meyer[1], Thomas Wies[2], and Sebastian Wolff[2](✉)

[1] TU Braunschweig, Braunschweig, Germany, meyer@tu-bs.de
[2] New York University, New York, USA, {wies,sebastian.wolff}@cs.nyu.edu

Abstract We present a new flow framework for separation logic reasoning about programs that manipulate general graphs. The framework overcomes problems in earlier developments: it is based on standard fixed point theory, guarantees least flows, rules out vanishing flows, and has an easy to understand notion of footprint as needed for soundness of the frame rule. In addition, we present algorithms for automating the frame rule, which we evaluate on graph updates extracted from linearizability proofs for concurrent data structures. The evaluation demonstrates that our algorithms help to automate key aspects of these proofs that have previously relied on user guidance or heuristics.

Keywords: Separation Logic · Graph Algorithms · Frame Inference.

1 Introduction

The flow framework [23,24] is an abstraction mechanism based on separation logic [5, 32, 40] that enables reasoning about global inductive invariants of general graphs in a local manner. The framework has proved useful to verify intricate algorithms that are difficult to handle by other techniques, such as the Priority Inheritance Protocol, object-oriented design patterns, and complex concurrent data structures [22,24,27,34]. However, these efforts have also exposed some rough corners in the underlying meta theory that either limit expressivity or automation. In this paper, we propose a new meta theory for the flow framework that aims to strike a balance between these conflicting requirements. In addition, we present algorithms that aid proof automation.

Background. The central notion of the flow framework is that of a *flow*. Given a commutative monoid $(\mathbb{M}, +, 0)$ (e.g. natural numbers with addition), and a graph with nodes X and an *edge function* $E\colon X \times X \to \mathbb{M} \to \mathbb{M}$, a flow is a function $\mathit{fl}\colon X \to \mathbb{M}$ that satisfies the *flow equation*:

$$\forall x \in X. \quad \mathit{fl}(x) = \mathit{in}_x + \textstyle\sum_{y \in X} E_{(y,x)}(\mathit{fl}(y)) \ .$$

That is, fl is a fixed point of the function that assigns every node x an initial value $\mathit{in}_x \in \mathbb{M}$, its *inflow*, and then propagates these values through the graph according to the edge function. This is akin to a forward data flow analysis where the monoid operation $+$ is used as the join. By choosing an appropriate flow monoid, inflow, and edge function, one can express inductive properties of graphs (reachability, sortedness, etc.) in terms of conditions that refer only to each node's flow value $\mathit{fl}(x)$.

A graph endowed with an inflow and associated flow is a *flow graph*. An example flow graph h is shown on the right-hand side of Fig. 1a. Here, the flow value $\mathit{fl}(w)$ for

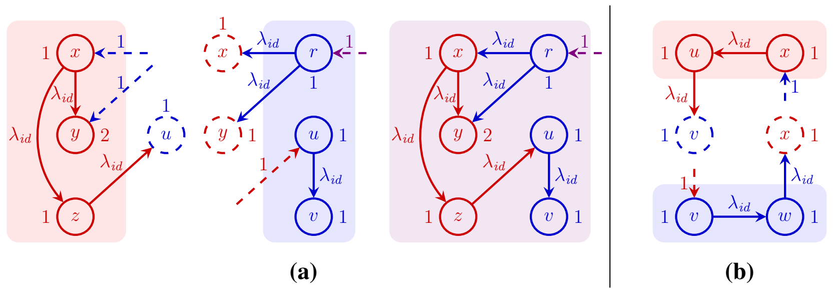

Figure 1. (a) Two flow graphs h_1 with nodes $h_1.X = \{ x, y, z \}$ (left) and h_2 with nodes $h_2.X = \{ r, u, v \}$ (center) for the flow monoid of natural numbers with addition. The edge label λ_{id} stands for the identity function. Omitted edges are labeled by the constant 0 function. Dashed edges represent the inflows. Nodes are labeled by their flow, respectively, outflow. The right side shows the composition $h = h_1 * h_2$. **(b)** Two flow graphs h_1 with $h_1.X = \{ u, x \}$ (top) and h_2 with $h_2.X = \{ v, w \}$ (bottom) whose composition is undefined due to vanishing flows.

a node w counts the number of paths from r to w. A flow graph can be partial and have edges to nodes outside of X like the node u for h_1 in Fig. 1a. If we include these nodes in the computation of the flow, then their flow values constitute the *outflow* of the flow graph. For instance, the outflow of h_1 for u is 1.

Flow graphs are equipped with a notion of disjoint composition, $h = h_1 * h_2$. An example is given in Fig. 1a. The composition is only defined if the union of the flows of h_1 and h_2 is again a flow of h. This may not always be the case. For instance, the inflows and outflows of h_1 and h_2 may be mutually incompatible such as h_1 sending outflow 2 to u whereas the inflow to u in h_2 is only 1.

Flow graph composition yields a *separation algebra*. That is, if we use flow graphs as an abstraction of program states (e.g., the heap), then we can use separation logic to reason locally about properties of programs that are expressed in terms of the induced flow graphs. For example, suppose the program updates the flow graph h in Fig. 1a to a new flow graph h' by inserting a new edge labeled λ_{id} between the nodes r and u. This increases the flow of u and v from 1 to 2. We can break this update down as follows. First, we decompose h into h_1 and h_2. Next, we obtain h_2' from h_2 by inserting the edge and updating the flow of u and v to 2. Finally, we compose h_2' again with h_1 to obtain h'. Note that the composition $h_1 * h_2'$ is still defined. This means that any property expressed over the flow in the h_1-portion of h still holds in h'. This is the well-known *frame rule* of separation logic, instantiated for flow graphs.

The crux in applying the frame rule is to show that the composition $h_1 * h_2'$ is indeed defined. One can do this locally by showing that the update $h_2 \rightsquigarrow h_2'$ is *frame-preserving*, i.e., for *any* h_1 such that $h_1 * h_2$ is defined, $h_1 * h_2'$ is also defined.

Typically, the flow subgraphs involved in a frame-preserving update $h_2 \rightsquigarrow h_2'$ include more nodes than those immediately affected by the update. For instance, consider the subgraphs of h and h' in our example that consist only of the nodes $\{r, u\}$ directly affected by inserting the edge. These subgraphs do not constitute a frame-preserving update because inserting the edge between r and u also changes the outflow to v from

1 to 2. Hence, the updated subgraph for $\{r, u\}$ would no longer compose with the rest of h where v's flow is still 1 instead of 2. We refer to a set of nodes such as $\{r, u, v\}$ that identifies a frame-preserving update as the update's *footprint*.

Meta theories of flow graphs. In addition to ensuring that flow graph composition yields a separation algebra, there are two desiderata that one has to take into consideration when designing a meta theory of flow graphs:

– *Obtaining unique flows.* When encoding inductive properties using flows, one is often interested in a particular flow, most commonly the least fixed point of the flow equation for a given inflow. One therefore needs a way to focus the reasoning on the particular flow of interest.
– *Identifying frame-preserving updates.* In order to enable the application of the frame rule, one needs a way to effectively compute candidate footprints and check whether they identify frame-preserving updates.

The first subgoal is crucial for expressivity and the second one for proof automation. Achieving one subgoals makes it more difficult to achieve the other. Specifically, consider the meta theory proposed in [24]. It requires that the flow monoid $(\mathbb{M}, +, 0)$ is also cancellative ($m + n_1 = o$ and $m + n_2 = o$ implies $n_1 = n_2$). Requiring cancellativity has the advantage that it is easy to check if an update $h \rightsquigarrow h'$ is frame-preserving: it suffices to show that h and h' have the same inflow and outflow. Cancellativity also ensures that for each flow fl, there exists a unique inflow that produces fl. Hence, it is sufficient to track only fl since the inflow is a derived quantity. However, the converse does not hold.

In fact, obtaining unique flows for cancellative $\mathbb{M}$ becomes more difficult. A natural requirement that one would like to impose on $\mathbb{M}$ is that the pre-order induced by $+$ forms a complete partial order (cpo) or even a complete lattice. This way, one can focus on the least flow, which is guaranteed to exist if one applies standard fixed point theorems, imposing only mild assumptions on the edge functions. However, cancellativity is inherently incompatible with standard domain-theoretic prerequisites. For instance, the only ordered cancellative commutative monoid that is a directed cpo is the trivial one: $\mathbb{M}_0 = \{0\}$. Similarly, $\mathbb{M}_0$ is the only such monoid that has a greatest element.

For cases where unique flows are desired, [24] imposes additional requirements on the edge functions (nil-potent) or the graph structure (effectively acyclic). The former is quite restrictive in terms of expressivity. The latter again complicates the computation of frame-preserving updates: one now has to ensure that no cycles are introduced when the updated graph h_2' is composed with its frame h_1. In fact, for the effectively acyclic case, [24] only provides a sufficient condition that a given footprint yields a frame-preserving update but it gives no algorithm for computing such a footprint.

Contributions. In this paper, we propose a new meta theory of flows based on flow monoids that form ω-cpos (but need not be cancellative). The cpo requirement yields the desired least fixed point semantics. The differences in the requirements on the flow monoid necessitate a new notion of flow graph composition. In particular, for a least fixed point semantics of flows, $h = h_1 * h_2$ is only defined if the flows of h_1 and h_2 do not vanish. An example of such a situation is shown in Fig. 1b, where the flows in h_1 and h_2 would vanish to 0 in $h_1 * h_2$ because the created cycle has no external inflow. Moreover, an update $h \rightsquigarrow h'$ is frame-preserving if h and h' route inflows to outflows in the same way. We formalize this condition using a notion of contextual equivalence of the graphs' *transfer functions*, which are the least fixed points of the flow equation, parameterized by the inflows and restricted to the nodes outside the graphs. We then identify conditions on the edge functions that are commonly satisfied in practice and that allow us to effectively check contextual equivalence of transfer functions. This result is remarkable because the flow monoid can have infinite ascending chains and the flow graphs can be cyclic. Building on this equivalence check, we propose an iterative algorithm for computing footprints of updates. This algorithm enables the automation of the frame rule for reasoning about programs manipulating flow graphs. We evaluate the presented algorithms on a benchmark suite of flow graph updates that are extracted from linearizability proofs for concurrent search structures constructed by the tool `plankton` [26,27]. The evaluation demonstrates that our algorithms help to automate key aspects of these proofs that have previously relied on user guidance or heuristics.

2 Flow Graph Separation Algebra

We start with the presentation of our new separation algebra of flow graphs.

Given a commutative monoid $(\mathbb{M}, +, 0)$, we define the binary relation $\leq$ on $\mathbb{M}$ by $n \leq m$ if there is $o \in \mathbb{M}$ with $m = n + o$. Flow values are drawn from a *flow monoid*, a commutative monoid for which the relation $\leq$ is an ω-cpo. That is, $\leq$ is a partial order and every ascending chain $K = m_0 \leq m_1 \leq \dots$ in $\mathbb{M}$ has a least upper bound, denoted $\bigsqcup K$. In the following, we fix a flow monoid $(\mathbb{M}, +, 0)$.

Let $\mathit{ContFun}(\mathbb{M} \to \mathbb{M})$ be the continuous functions in $\mathbb{M} \to \mathbb{M}$. Recall that a function $f : \mathbb{M} \to \mathbb{M}$ is *continuous* [43] if it commutes with limits of ascending chains, $f(\bigsqcup K) = \bigsqcup f(K)$ for every chain K in $\mathbb{M}$. We lift $+$ and $\leq$ to functions $\mathbb{M} \to \mathbb{M}$ in the expected way. An empty iterated sum $\sum_{i \in \varnothing} m_i$ is defined to be 0.

Lemma 1. *$(\mathit{ContFun}(\mathbb{M} \to \mathbb{M}), \circ, \mathit{id})$ is a monoid. Moreover, if $(\mathbb{M}, \leq)$ is an ω-cpo, so is $(\mathit{ContFun}(\mathbb{M} \to \mathbb{M}), \leq)$.*

A *flow graph* is a tuple $h = (X, E, \mathit{in})$ consisting of a finite set of nodes $X \subseteq \mathbb{N}$, a set of edges $E : X \times \mathbb{N} \to \mathit{ContFun}(\mathbb{M} \to \mathbb{M})$ labeled by continuous functions, and an *inflow* $\mathit{in} : (\mathbb{N} \setminus X) \times X \to \mathbb{M}$. We use FG for the set of all flow graphs and denote the empty flow graph by $h_\varnothing \triangleq (\varnothing, \varnothing, \varnothing)$.

We define two derived functions for flow graphs. First, the *flow* is the least function $\mathit{flow} : X \to \mathbb{M}$ satisfying the flow equation: $\mathit{flow}(x) = \mathit{in}_x + \mathit{rhs}_x(\mathit{flow})$, for all $x \in X$. Here, $\mathit{in}_x \triangleq \sum_{y \in (\mathbb{N} \setminus X)} \mathit{in}(y, x)$ is a monoid value and $\mathit{rhs}_x \triangleq \sum_{y \in X} E_{(y,x)}$ is a function of type $\mathit{ContFun}((X \to \mathbb{M}) \to \mathbb{M})$. Finally, we also define the *outflow* $\mathit{out} : X \times (\mathbb{N} \setminus X) \to \mathbb{M}$ by $\mathit{out}(x, y) \triangleq E_{(x,y)}(\mathit{flow}(x))$.

Example 1. For linearizability proofs of concurrent search structures one can use a flow that labels every data structure node x with its *inset*, the set of keys k' such that a thread searching for k' may traverse the node x [22,23]. Translated to our setting, the relevant flow monoid is the powerset of keys, $\mathbb{P}(\mathbb{Z} \cup \{ -\infty, \infty \})$, with set union as addition. Figure 2 shows two keyset flow graphs that abstract potential states of a concurrent set implementation based on sorted linked lists. When a key k is removed from the set, the node x that stores k is first marked to indicate that x has been logically deleted. In

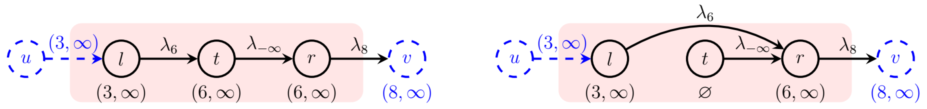

Figure 2. Two flow graphs h_1 (left) and h_2 (right) with $h_1.X = h_2.X = \{ l, t, r \}$ for the keyset flow monoid $\mathbb{P}(\mathbb{Z} \cup \{ -\infty, \infty \})$. The edge label λ_k for a key k denotes the function $\lambda m. \, (m \setminus [-\infty, k])$.

a second step, x is then physically unlinked from the list. The idea of the abstraction is that an edge leaving a node x that stores a key k is labeled by the function λ_k if x is unmarked and otherwise by $\lambda_{-\infty}$. This is because a search for $k' \in \mathbb{Z}$ will traverse the edge leaving x iff $k < k'$ or x is marked. In the figure, l and r are assumed to be unmarked, storing keys 6 and 8, respectively. Node t is assumed to be marked. Flow graph h_2 is obtained from h_1 by physically unlinking the marked node t. Using the keyset flow one can then express the crucial data structure invariants that are needed for a linearizability proof based on local reasoning (e.g., the invariant that the logical contents of a node is always a subset of its inset).

We note that the inflow of the global flow graph that abstracts the program state can be used in the specification. In the example, one lets $in_r = \mathbb{Z}$ for the root r of the data structure and $in_x = \varnothing$ for all other nodes to indicate that all searches start at r. □

Composition without vanishing flows. To define the composition of flow graphs, $h_1 * h_2$, we proceed in two steps. We first define an auxiliary composition that may suffer from *vanishing* flows, local flows that disappear in the composition. That is, this composition is defined for the flow graphs shown in Fig. 1b. In the composed graph the flow of each node is 0 where it was 1 before the composition—the flow vanishes. This means that the auxiliary composition does not allow to lift lower bounds on the flow values from the individual components to the composed graph. Hence, the actual composition restricts the auxiliary composition to rule out such vanishing flows. Definedness of the auxiliary composition requires disjointness of the nodes in h_1 and h_2. Moreover, the outflow of one flow graph has to match the inflow expectations of the other:

$$h_1 \#\# h_2 \quad \text{if} \quad X_1 \cap X_2 = \varnothing \ \wedge \ \forall x \in X_1, \, y \in X_2. \, out_1(x, y) = in_2(x, y) \wedge \\ out_2(y, x) = in_1(y, x) \; .$$

The auxiliary composition $h_1 \uplus h_2$ removes the inflow provided by the other component:

$$h_1 \uplus h_2 \quad \triangleq \quad (X_1 \uplus X_2, E_1 \uplus E_2, (in_1 \uplus in_2)|_{(\mathbb{N} \setminus (X_1 \uplus X_2)) \times (X_1 \uplus X_2)}) \; .$$

To rule out vanishing flows, we incorporate a suitable equality on the flows:

$$h_1 \# h_2 \quad \text{if} \quad h_1 \#\# h_2 \ \wedge \ h_1.\mathit{flow} \uplus h_2.\mathit{flow} = (h_1 \uplus h_2).\mathit{flow} \; .$$

Only if the latter equality holds, do we have the composition $h_1 * h_2 \triangleq h_1 \uplus h_2$. It is worth noting that $h_1.\mathit{flow} \uplus h_2.\mathit{flow} \geq (h_1 \uplus h_2).\mathit{flow}$ always holds. What definedness really asks for is the reverse inequality.

Recall from [5] that a *separation algebra* is a partial commutative monoid $(\Sigma, *, \mathsf{emp})$ with a set of units $\mathsf{emp} \subseteq \Sigma$.

Lemma 2. *$(FG, *, \{ h_\varnothing \})$ is a separation algebra.*

3 Frame-Preserving Updates

Since flow graphs form a separation algebra, we can use separation logic assertions to describe sets of flow graphs as in [24] and then use them to prove separation logic Hoare triples. A key proof rule used in such proofs is the frame rule. Given separation logic assertions P_1 and P_2, and a command c, the frame rule states: if the Hoare triple $\{P_1\}\, c\, \{P_2\}$ is valid, then so is $\{P_1 * F\}\, c\, \{P_2 * F\}$ for any *frame* F. The remainder of the paper focuses on developing algorithms for automating this proof rule.

The flow graphs described by an assertion may have unbounded size (e.g., due to the use of *iterated separating conjunctions*). We only consider bounded flow graphs in the following; the unbounded case is known to be a challenge for which orthogonal techniques are being developed (cf. Sect. 6). However, even if the flow graphs have bounded size, there may still be infinitely many of them because the inflows and edge functions are encoded symbolically in a logical theory of the flow monoid. For pedagogy, we present our algorithms in terms of concrete flow graphs rather than symbolic ones. However, our development readily extends to symbolic representations assuming the underlying flow monoid theory is decidable. In fact, our implementation discussed in Sect. 5 works with symbolic flow graphs.

The soundness of the frame rule relies on the assumption that the state update induced by the command c satisfies a certain locality condition. In our setting, this condition amounts to checking that the update of P_1 under c is *frame-preserving* with respect to flow graph composition. For the flow graphs h_1 described by P_1 and all flow graphs h_2 in the post image of h_1 under c, this means that $h_1 \# h$ implies $h_2 \# h$ for all h. Intuitively, $h_2 \# h$ still holds if h_1 and h_2 transfer inflows to outflows in the same way.

Formally, for a flow graph h we define its *transfer function* $\mathit{tf}(h)$ mapping inflows to outflows, $\mathit{tf}(h) : ((\mathbb{N} \setminus X) \times X \to \mathbb{M}) \to X \times (\mathbb{N} \setminus X) \to \mathbb{M}$, by

$$\mathit{tf}(h)(\mathit{in}') \triangleq h[\mathit{in} \mapsto \mathit{in}'].\mathit{out}\ .$$

For a given inflow in, we also write $\mathit{tf}(h_1) =_{\mathit{in}} \mathit{tf}(h_2)$ to mean that for all inflows $\mathit{in}' \leq \mathit{in}$, $\mathit{tf}(h_1)(\mathit{in}') = \mathit{tf}(h_2)(\mathit{in}')$.

Definition 1. *Flow graphs* h_1, h_2 *are* contextually equivalent*, denoted* $h_1 =_{\mathit{ctx}} h_2$*, if we have* $h_1.X = h_2.X$*,* $h_1.\mathit{in} = h_2.\mathit{in}$*, and* $\mathit{tf}(h_1) =_{h_1.\mathit{in}} \mathit{tf}(h_2)$*.*

Theorem 1 (Frame Preservation). *For all flow graphs* $h_1 =_{\mathit{ctx}} h_2$ *and* h*,* $h_1 \# h$ *if and only if* $h_2 \# h$ *and, in case of definedness,* $h_1 * h =_{\mathit{ctx}} h_2 * h$*.*

To automate the frame rule for a command c and a precondition P, we need to identify a decomposition $P = P_1 * F$ so as to infer $\{P_1\}\, c\, \{P_2\}$ and then apply the frame rule to derive $\{P\}\, c\, \{Q\}$ for the postcondition $Q = P_2 * F$. This is closely related to the *frame inference problem* [4]. When a command modifies a flow graph h_1 to h_2, our goal is to identify a (hopefully small) set of nodes Y in h_1 that are affected by this update, the *flow footprint*. That is, Y captures the difference between the flow graphs before and after the update and the complement of Y defines the frame. To make this formal, we need the restriction of flow graphs to subsets of nodes, which then gives us a notion of flow graph decomposition. Towards this, consider h and $Y \subseteq \mathbb{N}$. We define

$$h|_Y \;\triangleq\; (h.X \cap Y, h.E|_{(h.X \cap Y) \times \mathbb{N}}, \mathit{in})$$

such that the inflow in satisfies $in(z,y) \triangleq h.in(z,y)$ for all $z \in \mathbb{N} \setminus h.X$, $y \in h.X \cap Y$ and $in(x,y) \triangleq h.E_{(x,y)}(h.\mathit{flow}(x))$ for all $x \in h.X \setminus Y$, $y \in h.X \cap Y$.

Definition 2. *Consider h_1 and h_2 with $X \triangleq h_1.X = h_2.X$ and $h_1.in = h_2.in$. A* flow footprint *for the difference between h_1 and h_2 is a subset of nodes $Y \subseteq X$ so that $h_1|_Y =_{ctx} h_2|_Y$ and $h_1|_{X\setminus Y} = h_2|_{X\setminus Y}$. The set of all such footprints is $\mathit{FFP}(h_1,h_2)$.*

Flow graphs over different sets of nodes or inflows never have a flow footprint. The former requirement merely simplifies the presentation. To that end, we assume that all nodes that will be allocated during program execution are already present in the initial flow graph. This assumption can be lifted. The latter requirement is motivated by the fact that the global inflow is part of the specification as noted earlier in Example 1.

Before we proceed with the problem of how to compute flow footprints, we highlight some of their properties.

Lemma 3 (Footprint Monotonicity). *If $Z \in \mathit{FFP}(h_1,h_2)$ and $Z \subseteq Y \subseteq h_1.X$, then $Y \in \mathit{FFP}(h_1,h_2)$.*

A consequence of monotonicity is the existence of a canonical flow footprint: if there is a flow footprint at all, then the set of all nodes will work as a footprint. Of course this canonical footprint is undesirably large. It corresponds to the case where one reasons about flow graph updates globally, forgoing the application of the frame rule. Unfortunately, an inclusion-minimal flow footprint does not exist.

Proposition 1 (Canonical Footprints). *We have: $\mathit{FFP}(h_1,h_2) \neq \varnothing$ if and only if $h_1.X \in \mathit{FFP}(h_1,h_2)$. There is no inclusion-minimal flow footprint; in particular, the set $\mathit{FFP}(h_1,h_2)$ is not closed under intersection.*

The proof of monotonicity requires a better understanding of the restriction operator, as provided by the following lemma.

Lemma 4 (Restriction). *Consider h and $Y, Z \subseteq \mathbb{N}$. Then (i) $h|_Y.\mathit{flow} = h.\mathit{flow}|_Y$, (ii) $h|_Y \# h|_{X\setminus Y}$ and $h|_Y * h|_{X\setminus Y} = h$, and (iii) $(h|_Y)|_Z = h|_{Y\cap Z}$.*

Since flow footprints are defined via restriction, the lemma also shows that flow footprints are well-behaved. For example, the restriction to the footprint Y does not change the flow of a node $y \in Y$ nor that of a node $x \in h.X \setminus Y$. More formally, this means $h|_Y.\mathit{flow}(y) = h.\mathit{flow}(y)$ and $h|_{X\setminus Y}.\mathit{flow}(x) = h.\mathit{flow}(x)$, by Lemma 4(i).

For our development, it will be convenient to have a more operational formulation of the transfer function. Towards this, we understand the flow graph as a function that takes an inflow as a parameter and yields a transformer of flow approximants:

$$\begin{array}{ll} & h \;:\; ((\mathbb{N}\setminus X)\times X \to \mathbb{M}) \to (X \to \mathbb{M}) \to X \to \mathbb{M} \\ \text{defined by} & h[in](\sigma)(x) \;=\; in_x + \mathit{rhs}_x(\sigma)\ . \end{array}$$

Recall $in_x \triangleq \sum_{y\in\mathbb{N}\setminus X} in(y,x)$ and $\mathit{rhs}_x(\sigma) = \sum_{y \in X} E_{(y,x)}(\sigma(y))$. The least fixed point of $h[in]$ is $\bigsqcup_{i\in\mathbb{N}} h[in]^i(\bot)$ with $h^0 = id_{X\to\mathbb{M}}$ and $h^{i+1} = h^i \circ h$, by Kleene's theorem. Define $out : (X \to \mathbb{M}) \to X \times (\mathbb{N}\setminus X) \to \mathbb{M}$ by $out(\sigma)(y,z) \triangleq E_{(y,z)}(\sigma(y))$. This yields the following characterization of transfer functions and flows.

Lemma 5 (Transfer). *For all flow graphs h we have (i) $\mathit{tf}(h) = out \circ (\mathit{lfp}.h[-])$ and (ii) $\mathit{lfp}.h[h.in]) = h.\mathit{flow}$.*

4 Computing Footprints

We present an algorithm for computing a footprint for the difference between two given flow graphs. We proceed in two steps. We first give a high-level description of the algorithm that ignores computability problems. In a second step, we show how to solve the computability problems. Throughout the development, we will assume to have flow graphs h_1 and h_2 over the same nodes $X \triangleq h_1.X = h_2.X$ and with the same inflow $h_1.in = h_2.in$. If this assumption fails, a flow footprint does not exist by definition.

4.1 Algorithm

We compute the flow footprint as a fixed point. We start with the footprint candidate Z consisting of the nodes whose outgoing edges differ in h_1 and h_2. Then, we iteratively add the nodes whose outflow leaving the current footprint candidate Z differs in $h_1|_Z$ and $h_2|_Z$. That the outflow differs means that the transfer functions $\mathit{tf}(h_1|_Z)$ and $\mathit{tf}(h_2|_Z)$ differ and thus the candidate Z is not a footprint. In turn, if all outflows match, the transfer functions coincide and Z is a footprint as desired.

Technically, we compute the fixed point over the powerset lattice of nodes endowed with a distinguished top element: $(\mathbb{P}(X)^\top, \sqsubseteq)$ with $\mathbb{P}(X)^\top \triangleq \mathbb{P}(X) \uplus \{\top\}$. Element $\top$ indicates a failure of the footprint computation. This may arise if the footprint is not covered by X, i.e., extends beyond the flow graphs h_1, h_2.

Our fixed point computation starts from $Z = \mathit{odif}_{h_1,h_2} \subseteq X$ as defined by

$$\mathit{odif}_{h_1,h_2} \triangleq \{ x \in X \mid \exists z \in \mathbb{N}. h_1.E(x,z) \neq h_2.E(x,z) \} .$$

The fixed point then proceeds to extend Z as long as the transfer functions associated with $h_1|_Z$ and $h_2|_Z$ do not match. To define the extension, we let the *transfer failure* of $Z \subseteq X$ be the successor nodes of Z that may receive different outflow from h_1 and h_2:

$$\mathit{tfail}_{h_1,h_2}(Z) \triangleq \left\{ x \in \mathbb{N} \setminus Z \;\middle|\; \begin{array}{l} \exists\, in \leq h_1|_Z.in \;\; \exists\, z \in Z. \\ \qquad [\mathit{tf}(h_1|_Z)(in)](z,x) \neq [\mathit{tf}(h_2|_Z)(in)](z,x) \end{array} \right\} .$$

This set is the *reason* why the current footprint candidate Z is not a footprint, that is, $Z \notin \mathit{FFP}(h_1,h_2)$. Extending Z with the transfer failure yields a new candidate. We check that the new candidate is covered by X (i.e., does not include nodes outside of h_1, h_2). If the check fails, the new candidate is $\{\top\}$ to indicate that no footprint could be computed. The following definition makes the extension procedure precise.

Definition 3. *The function* $\mathit{ext}_{h_1,h_2} : \mathbb{P}(X)^\top \to \mathbb{P}(X)^\top$ *is defined by*

$$\mathit{ext}_{h_1,h_2}(Z) \triangleq \mathit{tfail}_{h_1,h_2}(Z) \not\subseteq X \;\textbf{?}\; \top \;\textbf{:}\; Z \sqcup \mathit{odif}_{h_1,h_2} \sqcup \mathit{tfail}_{h_1,h_2}(Z) .$$

Iteratively extending the candidate Z with the transfer failure eventually produces a footprint for the difference of h_1 and h_2, or fails with $\top$. The approach is sound.

Theorem 2 (Soundness). *Let* $F \triangleq \mathit{lfp}.\mathit{ext}_{h_1,h_2}$. *If* $F \neq \top$, *then* $F \in \mathit{FFP}(h_1,h_2)$.

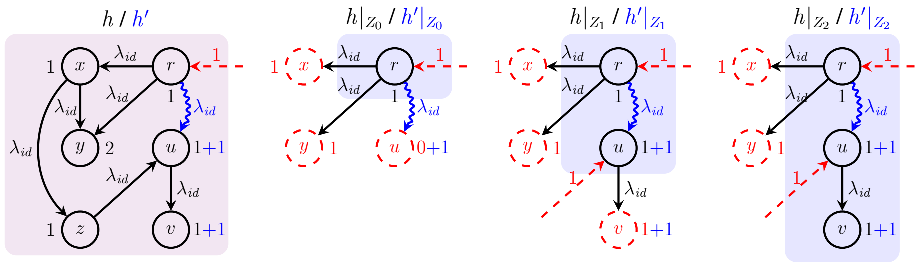

Figure 3. Computing a footprint for the difference of h and h' iterates through the sets $Z_0 \triangleq \{ r \}$, $Z_1 \triangleq \{ r, u \}$, and $Z_2 \triangleq \{ r, u, v \}$. The latter is the least fixed point of $ext_{h,h'}$ and a footprint as desired, $Z_2 \in FFP(h, h')$.

Example 2. For an illustration consider Fig. 3. There, we apply the fixed point computation to find a footprint for the difference of h and h'. As alluded to in Sect. 1, h' is the result of inserting into h a new edge between nodes r and u labeled with λ_{id}.

The fixed point computation starts from $Z_0 \triangleq \{ r \} = odif_{H,H'}$ as it is the only node whose outgoing edges have changed. Next, we compute $tfail_{h,h'}(Z_0)$. This yields $\{ u \}$ because u receives 0 from Z_0 in h but 1 in h' due to the new edge. The outflow from Z_0 to the remaining nodes coincides in h and h'. Hence, the extension of Z_0 with the transfer failure yields $Z_1 \triangleq ext_{h,h'}(Z_0) = \{ u, r \}$. Similarly, we compute $tfail_{h,h'}(Z_1)$ and obtain $Z_2 \triangleq ext_{h,h'}(Z_1) = \{ r, u, v \}$. Since v has no outgoing edges, Z_2 is the least fixed point of $ext_{h,h'}$. Because Z_2 is a subset of the nodes of h and h', it is a footprint, $Z_2 \in FFP(h, h')$. □

To obtain Theorem 2, we have to prove that the fixed point $F \triangleq lfp.ext_{h_1,h_2}$ is indeed a footprint if $F \neq \top$. That is, we have to establish the following two properties according to Definition 2: (i) $h_1|_F =_{ctx} h_2|_F$ and (ii) $h_1|_{X \setminus F} = h_2|_{X \setminus F}$.

To see the latter one, note that the graph structures (the nodes and edges) of $h_1|_{X \setminus F}$ and $h_2|_{X \setminus F}$ coincide because $odif_{h_1,h_2} \subseteq F$. The inflows coincide as well because they are, intuitively, comprised of the flow graph's overall inflow $h_1.in = h_2.in$ and the outflow of the footprint, which is equal in both flow graphs due to $h_1|_F =_{ctx} h_2|_F$.

The interesting part of the soundness proof is to establish property (i), the contextual equivalence $h_1|_F =_{ctx} h_2|_F$. Since F is a fixed point of ext_{h_1,h_2}, we know that $tfail_{h_1,h_2}(Z) = \varnothing$ and thus the transfer functions of $h_1|_F$ and $h_2|_F$ coincide. Hence, it suffices to establish $h_1|_F.in = h_2|_F.in$ to obtain the desired contextual equivalence, Definition 1. This key step in the proof is obtained with the help of the following lemma.

Lemma 6. *Let $odif_{h_1,h_2} \subseteq F \subseteq X$ with $tfail_{h_1,h_2}(F) = \varnothing$. Then $h_1|_F.in = h_2|_F.in$.*

To establish the lemma one has to show that the inflow into F from the non-footprint part $Y \triangleq X \setminus F$ coincides in h_1 and h_2. The challenge is a cyclic dependency in the flow: the inflow from Y depends on the outflow of F, which depends on the inflow from Y. To tackle this, we rephrase the flow equation for h_i as a pairing of the two separate flow equations for $h_i|_F$ and $h_i|_Y$, for $i \in \{ 1, 2 \}$. Intuitively, the pairings compute the flow locally in $h_i|_F$ and $h_i|_Y$ for a fixed inflow (initially $h_i.in$). Then, the inflow to $h_i|_F$

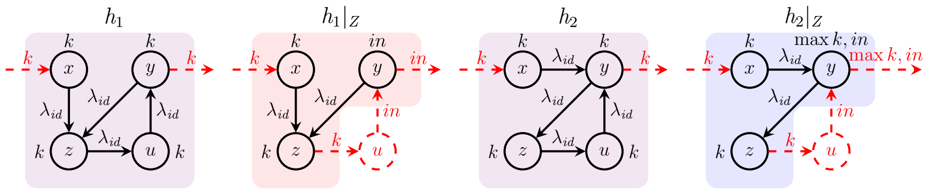

Figure 4. Counterexample to completeness using the monoid $(\mathbb{N}\cup\{\infty\}, \max, 0)$. While the set $\{x, y, z, u\}$ is a footprint for the difference between flow graphs h_1 and h_2, our fixed point will produce the candidates $\{x\}$ and $Z \triangleq \{x, y, z\}$ and then fail with $\{\top\}$.

is updated to the inflow from outside h_i and the inflow from $h_i|_Y$, and similarly for the inflow to $h_i|_Y$. This is repeated until a fixed point is reach. Technically, we rely on Bekić's Lemma [1] to compute the pairings. Then, we observe $\mathit{tf}(h_1|_F) = \mathit{tf}(h_2|_F)$ because $\mathit{tfail}_{h_1,h_2}(F) = \varnothing$ as well as $\mathit{tf}(h_1|_Y) = \mathit{tf}(h_2|_Y)$ because $\mathit{odif}_{h_1,h_2} \subseteq F$. Roughly, this means that the flow pairings for h_1 and h_2 must coincide as the individual parts propagate the same values. Put differently, the updated inflow for $h_1|_F$ and $h_2|_F$ as well as $h_1|_Y$ and $h_2|_Y$ coincide in each iteration. Overall, we get $h_1|_F.\mathit{in} = h_2|_F.\mathit{in}$.

Our computation of a flow footprint is forward, it starts from the nodes where the flow graphs differ and follows the edges. It may therefore fail if predecessor nodes of an iterate Z need to be considered to determine a flow footprint. For an example refer to Fig. 4. Using the monoid $(\mathbb{N}\cup\{\infty\}, \max, 0)$, it is easy to see that the set $\{\, x, y, z, u \,\}$ is a footprint for the difference between h_1 and h_2. Our fixed point, however, will start with $\{\, x \,\}$ and extend this to $Z \triangleq \{\, x, y, z \,\}$. Let v be the node outside the flow graphs that y is pointing to. Then, the next transfer failure is $\mathit{tfail}_{h_1,h_2}(Z) = \{\, v \,\}$ because for $\mathit{in} < k$ the outflow of y to v differs in $h_1|_Z$ and $h_2|_Z$. Our approach fails to compute a footprint.

Fact 3 (Incompleteness) *There are flow graphs h_1 and h_2 for which our algorithm is not able to determine a flow footprint although one exists.*

4.2 Comparing Transfer Functions

When implementing the above fixed point computation, the challenge is to prove the equivalence between given transfer functions in order to obtain the transfer failure: $[\mathit{tf}(h_1|_Z)(-)](-, x) = [\mathit{tf}(h_2|_Z)(-)](-, x)$? Already the comparison of two functions is known to be difficult to do algorithmically. What adds to the problem is that transfer functions are defined as least fixed points, meaning we do not have a closed-form representation of the functions to compare.

Our approach is to impose additional requirements on the set of edge functions. The requirements are met in all our experiments, and so do not mean a limitation for the applicability of our approach. We show that if the edge functions are not only continuous but also distributive, then the transfer functions can be understood in terms of paths through the underlying flow graphs. If the edge functions are additionally decreasing and the underlying monoid's addition is idempotent, then acyclic paths are sufficient. Both results do not hold for merely continuous edge functions.

Distributivity. Our first additional assumption is that the edge functions $f : \mathbb{M} \to \mathbb{M}$ are not only continuous, but also *distributive* in that $f(m+n) = f(m) + f(n)$ for all $m, n \in \mathbb{M}$ and $f(0) = 0$. We use $\mathit{DistFun}(\mathbb{M})$ to refer to the set of all continuous and distributive functions over $\mathbb{M}$. The properties formulated in Lemma 1 carry over.

For continuous and distributive transfer functions, we can understand $h[\mathit{in}]^i$ in terms of the paths through $h[\mathit{in}]$ of length i. For example, $i = 3$ yields

$$\begin{aligned}[h[\mathit{in}]^3](\bot)(z) &= \mathit{in}_z + \sum_{y\in X} E_{(y,z)}(\ \mathit{in}_y + \sum_{x\in X} E_{(x,y)}(\mathit{in}_x + \sum_{u\in X} E_{(u,x)}(\bot(u))\ ) \\ &= \mathit{in}_z + \sum_{y\in X} E_{(y,z)}(\mathit{in}_y) + \sum_{y\in X}\sum_{x\in X} E_{(y,z)}(E_{(x,y)}(\mathit{in}_x))\ .\end{aligned}$$

The first equality is by definition, the second is where distributivity comes in. In particular, $\bot(u) = 0$ and so $E_{(y,z)}(\ E_{(x,y)}(\ E_{(u,x)}(\ \bot(u)\ )\ ) = 0$. The last term shows that we forward the inflow given at a node x to an intermediary node y and from there to the node z of interest. For higher powers of $h[\mathit{in}]$, we take longer paths. For $h[\mathit{in}]^*$, we thus obtain the sum over all nodes x and all paths from x to z through the flow graph. We need some definitions to make this precise.

A *path* p through flow graph h is a finite, non-empty sequence of nodes all of which belong to the flow graph except the last which lies outside:

$$p \ = \ x_0 \cdot \ldots \cdot x_n \cdot z \ \in \ X^+ \cdot (\mathbb{N} \setminus X)$$

where $\cdot$ denotes path concatenation. We use $\mathit{first}(p) = x_0$ resp. $\mathit{last}(p) = x_n$ to extract the first resp. last node from within the flow graph h. By $\mathit{Paths}(h, x, y, z)$ we denote the set of all paths through flow graph h that start in node $\mathit{first}(p) = x$ and leave h from node $\mathit{last}(p) = y$ to move to $z \in \mathbb{N} \setminus X$. Given a set of nodes $X' \subseteq X$, we use $\mathit{Paths}(h, X', y, z)$ for the union over all $x \in X'$ of the sets $\mathit{Paths}(h, x, y, z)$. The path induces the function $E_p : \mathbb{M} \to \mathbb{M}$ that composes the edge functions along the path:

$$E_x \ = \ \mathit{id} \qquad\qquad E_{x.p} \ = \ E_p \circ E_{(x,\mathit{first}(p))}\ .$$

Together with Lemma 5, the above analysis yields the first closed-form representation of a flow graph's transfer function, which so far has involved a fixed point computation.

Theorem 4 (Closed-Form Representation). *If h is labeled over $\mathit{DistFun}(\mathbb{M})$, then:*

$$[\mathit{tf}(h)(\mathit{in})](y,z) \quad = \quad \textstyle\sum_{x\,\in\, X}\ \sum_{p\,\in\, \mathit{Paths}(h,x,y,z)}\ E_p(\mathit{in}_x)\ .$$

Theorem 4 pushes the fixed point computation of transfer functions into the sets $\mathit{Paths}(h, x, y, z)$ which are themselves defined inductively and potentially infinite. In the following, we alleviate this problem without requiring acyclicity of the flow graph.

Idempotence. Our second assumption is that addition in the monoid is idempotent, meaning $m + m = m$ for all $m \in \mathbb{M}$. Idempotence ensures the addition degenerates to a join for comparable elements: $m+n = m \sqcup n = n$ for all $m \leq n \in \mathbb{M}$. Unless stated otherwise, we hereafter assume an idempotent addition.

With Theorem 4, it remains to compare sums over paths. With idempotence, we show that we can further reduce the problem and reason over single paths rather than sums. We show that every path in h_1 can be replaced by a set of paths in h_2, and vice versa. Even more, we only have to consider the paths from nodes where the edges changed. The precise formulation of the path replacement condition is the following.

Definition 4. *The* path replacement condition *for flow graphs h_1 by h_2 over the same set of nodes X and labeled by $\mathit{DistDecFun}(\mathbb{M})$ requires that for every $x \in \mathit{odif}_{h_1,h_2}$, for every $y \in X$, and for every $z \in \mathbb{N} \setminus X$ we have*

$$\forall\, p \in \mathit{Paths}(h_1, x, y, z)\ \ \exists\, P \subseteq \mathit{Paths}(h_2, x, y, z).\ \ E_p \ \leq\ E_P \ \triangleq\ \textstyle\sum_{q \in P} E_q\ .$$

Example 3. For the flow graphs h_1 and h_2 from Fig. 4, we have path replacement of h_1 by h_2, and vice versa. To see this, consider the path $p \triangleq x \cdot z \cdot u \cdot y \cdot v$ in h_1 and $q \triangleq x \cdot y \cdot v$ in h_2, where v is the node outside of h_1, h_2 that y points to. Since all edges are labeled with λ_{id}, we have $E_p = \lambda_{id} = E_q$. It is worth noting that, in this example, we can ignore the cycles in h_1 and h_2. In a moment, we will introduce restrictions on edge functions in order to do avoid cycles in general.

Similarly, we have path replacement for the flow graphs from Fig. 2. To be precise, $E_p = \lambda_8 = E_q$ for the paths $p \triangleq l \cdot t \cdot r \cdot v$ in h_1 and $q \triangleq l \cdot r \cdot v$ in h_2. $\square$

The main result is that path replacement is sound and complete for proving equivalence of transfer functions.

Theorem 5 (Path Replacement Principle). *We have $\mathit{tf}(h_1) = \mathit{tf}(h_2)$ if and only if path replacement of h_1 by h_2 and of h_2 by h_1 hold.*

The theorem is remarkable in several respects. First, one would expect we have to replace the paths from all nodes in h_1. Instead, we can focus on the nodes where the outgoing edges changed. Second, one would expect the replacing paths P start from arbitrary nodes in h_2. Such a set of paths would yield a transfer function of type $(Y \to \mathbb{M}) \to \mathbb{M}$. Instead, we can work with a function of type $\mathbb{M} \to \mathbb{M}$. Even more, we can focus on paths starting in the same node as the path we intend to replace. Finally, the paths we use for replacement come without any constraints, leaving room for heuristics.

The proof starts from a *full path replacement condition* of h_1 by h_2, both over X and labeled by $\mathit{DistFun}(\mathbb{M})$. Full path replacement coincides with Definition 4 but draws x from full X rather than $x \in \mathit{odif}_{h_1,h_2}$. Full path replacement characterizes equivalence of the transfer functions in a monoid with idempotent addition in the case of continuous and distributive edge functions.

Lemma 7. *Full path replacement of h_1 by h_2 and h_2 by h_1 hold iff $\mathit{tf}(h_1) = \mathit{tf}(h_2)$.*

The result is a consequence of Theorem 4, which equates $\mathit{tf}(h_1)$ with the sum of the E_p for all paths $p \in \mathit{Paths}(h_1, x, y, z)$ for all $x \in X$. Full path replacement allows us to sum over E_P instead, for some $P \subseteq \mathit{Paths}(h_2, x, y, z)$. Over-approximating P with all paths $\mathit{Paths}(h_2, x, y, z)$, we obtain an upper bound for $\mathit{tf}(h_1)$. It is easy to see that the resulting sum can be rewritten into the form of Theorem 4, yielding $\mathit{tf}(h_1) \leq \mathit{tf}(h_2)$. Analogously, we get $\mathit{tf}(h_1) \geq \mathit{tf}(h_2)$ and thus $\mathit{tf}(h_1) = \mathit{tf}(h_2)$ as required. The reverse direction of the lemma is similar.

To conclude the proof of the path replacement principle in Theorem 5, we show that full path replacement and (ordinary) path replacement of h_1 by h_2 coincide. To see this, consider a path $p \in \mathit{Paths}(h_1, x, y, z)$ for any $x \in X$. The goal is to show $E_p \leq E_P$ for some $P \in \mathit{Paths}(h_2, x, y, z)$. To that end, decompose the path into $p = p_1 \cdot p_2$ such that $x' \triangleq \mathit{first}(p_2)$ is the first node in p from odif_{h_1,h_2}. Ordinary path replacement yields $Q \in \mathit{Paths}(h_2, x', y, z)$ with $E_{p_2} \leq E_Q$. Now, choose $P \triangleq \{\, p_1 \cdot q \mid q \in Q \,\}$. Because p_1 exists in h_1 and h_2 with the exact same edge labels, we obtain the desired $E_p \leq E_P$.

Lemma 8. *Full path replacement of h_1 by h_2 holds if and only if path replacement of h_1 by h_2 holds.*

Decreasingness. We assume that the edge functions $f : \mathbb{M} \to \mathbb{M}$ are not only continuous and distributive, but also *decreasing*: $f(m) \leq m$ for all $m \in \mathbb{M}$. The assumption of decreasing edge functions is justified by the fact that a program that traverses the flow graph builds up information about the status of the structure, and smaller flow values mean more information (as in classical data flow analysis). We use $\mathit{DistDecFun}(\mathbb{M})$ to refer to the set of all continuous, distributive, and decreasing transfer functions over $\mathbb{M}$; Lemma 1 carries over to this set. Addition in the monoid is still assumed idempotent.

If all edge functions are decreasing, every cycle in the flow graph is decreasing as well. The key observation is that, given an idempotent addition, cycles with decreasing edge functions can be avoided when forming sums over sets of paths.

Lemma 9. *Let h be labeled over $\mathit{DistDecFun}(\mathbb{M})$ and $p_1 \cdot p \cdot p_2 \in \mathit{Paths}(h, x, y, z)$ with $\mathit{last}(p) = \mathit{first}(p)$. Then $p_1 \cdot p_2 \in \mathit{Paths}(h, x, y, z)$ and $E_{p_1 \cdot p \cdot p_2} \leq E_{p_1 \cdot p_2}$.*

Call a path *simple* if it does not repeat a node and let $\mathit{SimplePaths}(h, x, y, z)$ denote the set of all simple paths through h from x to y and leaving the flow graph towards z. Note that a finite graph only admits finitely many simple paths.

Theorem 6 (Simple Paths). *Assuming continuous, distributive, and decreasing edge functions, and assuming idempotent addition, Theorem 4 and Theorem 5 hold with every occurrency of $\mathit{Paths}(h, x, y, z)$ replaced by $\mathit{SimplePaths}(h, x, y, z)$.*

In practice, path-counting flows, keyset flows, reachability flows, shortest-path flows, and priority inheritance flows are relevant [22–24, 27] and compatible with our theory.

5 Evaluation

We substantiate the practicality of our new approach by evaluating it on a real-world collection of flow graphs extracted from the literature. We explain how we obtained our benchmarks and how we implemented and evaluated our approach.

Benchmark Suite. As alluded to in Sect. 1, the flow framework has been used to verify complex concurrent data structures. More specifically, it has been used for automated proof construction by the `plankton` tool [26, 27]. `plankton` performs an exhaustive proof search over a separation logic with support for flows—and further advanced features for establishing linearizability that do not matter for the present evaluation. In order to handle heap updates, `plankton` generates a footprint h for the flow graph $h_1 = h * h_{\mathit{frame}}$ of the current proof state (represented as an assertion in separation logic). It then frames the non-footprint part h_{frame} of the flow graph h_1 to compute the post state h' of the heap update locally for the footprint h. The result is the new flow graph $h_2 = h' * h_{frame}$. We consider the pair (h_1, h_2) a *benchmark* for our evaluation.

We adapt `plankton` to export the flow graph pairs for which a footprint is constructed. This way, we obtain 1272 benchmarks from the heap updates occurring during proof construction for a collection of 10 concurrent set data structures. All flow graphs in this benchmark suite contain at most 4 nodes.

Our benchmark suite is limited by the capabilities and restrictions of `plankton`. In particular, we inherit the confinement to concurrent search structures. This is due to the fact that `plankton` integrates support only for the keyset flow (cf. Example 1). Our evaluation will compute footprints with respect to this flow.

Implementation. We implement the fixed point computation to find footprints for two given flow graphs h_1, h_2 from Sect. 4 in a tool called `krill` [28]. It integrates three methods for computing the transfer failure $\mathit{tfail}_{h_1,h_2}(Z)$ of a footprint candidate Z:

1. **NAIVE:** A naive method that computes the flow within the footprint Z. Following [24], we require acyclicity of flow graphs for this method to avoid solving a fixed point equation when computing the flow.
2. **NEW:** Our new approach leveraging the path replacement condition (cf. Theorem 5) for simple paths (cf. Theorem 6). This method requires distributive and decreasing edge functions as well as idempotent addition in the underlying monoid.
3. **DIST:** A variation of our new approach leveraging the closed-form representation (cf. Theorem 4). We require distributive edge functions and acyclicity of the flow graphs to avoid an unbounded sum over all paths in the closed-form representation.

Our benchmark suite satisfies the requirements for all three methods. The NAIVE and DIST methods include a (sufficient) check to ensure acyclicity in the updated flow graph to guarantee soundness of the resulting footprint.

All three methods encode the necessary equivalence checks among transfer functions as SMT formulas which are then discharged using the off-the-shelf SMT solver `Z3` [31]. Our encodings use the theory of integers with quantifiers. The NAIVE method additionally uses free functions to encode sets of integers.

Experiments. We ran `krill` on our benchmark suite and compared the runtime of the three different methods for computing the transfer failure. Our results are summarized in Fig. 5(left). For every search structure that we extracted benchmarks from, the figure lists: (i) the number #FG of flow graph pairs extracted, (ii) each method's total runtime for computing the footprints of all flow graph pairs, and (iii) the speedup of NEW over NAIVE in percent. The experiments were conducted on an Apple M1 Pro.

Figure 5(left) shows that the runtime for all methods is roughly linear in the number of computed footprints. Moreover, the absolute time for computing footprints is small, making the approaches practical. The figure also shows that our NEW and DIST methods have a performance advantage over the NAIVE method. The NEW method is between 22% and 39% faster than the NAIVE method. We believe that the difference is relatively small only because the acyclicity assumption avoids a potentially non-terminating fixed point computation. Avoiding this fixed point in the presence of cycles is a major advantage that our NEW method has over the NAIVE and DIST methods. The performance difference for DIST and NEW are negligible because the acyclicity check is negligible.

Structure	#FG	NAIVE	DIST	NEW	Speedup
Fine set [13]	12	75 ms	48 ms	46 ms	39%
Lazy set [12]	14	73 ms	52 ms	51 ms	30%
ORVYY set [33]	20	106 ms	76 ms	74 ms	30%
VY DCAS set [46]	19	109 ms	74 ms	73 ms	33%
VY CAS set [46]	28	139 ms	104 ms	102 ms	27%
Michael set [29]	225	1216 ms	887 ms	874 ms	28%
Michael set (wait-free)	186	996 ms	731 ms	721 ms	27%
Harris set [11]	352	2242 ms	1490 ms	1443 ms	36%
Harris set (wait-free)	296	1859 ms	1242 ms	1205 ms	35%
FEMRS tree [10]	120	519 ms	409 ms	407 ms	22%
Total	1272	7335 ms	5114 ms	4996 ms	32%

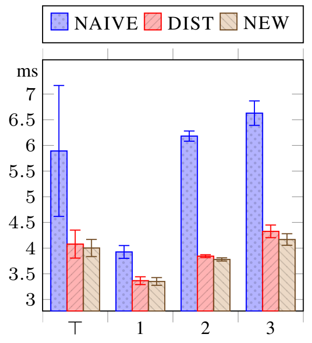

Figure 5. Experimental results averaged over 1000 repeated runs, conducted on an Apple M1 Pro. **(left)** Total runtime for computing footprints for flow graphs occurring during automated proof construction for highly concurrent set data structures. The speedup gives the relative performance improvement of NEW over NAIVE. **(right)** Average runtime for computing a single footprint, partitioned by footprint size ($\top$ indicates failure).

We also factorized the runtimes of our benchmarks along the size of the resulting footprint. Figure 5(right) gives the average runtime and standard deviation for computing a single footprint, broken down by footprint size. If no footprint could be found, its size is listed as $\top$. These failed footprint constructions are consistent with `plankton`'s method and would not lead to verification failure.

6 Related Work

Two alternative meta theories for the flow framework have been proposed in prior work [23, 24]. Like in our setup, the original flow framework [23] demands that the flow domain is an ω-cpo to obtain a least fixed point semantics. However, it proposes a different flow graph composition that leads to a notion of contextual equivalence relying on inflow equivalence classes. This complicates proof automation. In addition, the flow domain is assumed to be a semiring and edge functions are restricted to multiplication with a constant. This limits expressivity.

As discussed in Sect. 1, the revised flow framework proposed in [24] requires that the flow monoid is cancellative but not an ω-cpo. This means that uniqueness of flows is not guaranteed per se. Instead, uniqueness is obtained by imposing additional conditions on the edge functions. However, these conditions are more restrictive than those imposed in our framework. The *capacity* of a flow graph introduced in [24] closely relates to our notion of transfer function. A closed-form representation based on sums over paths is used to check equivalence of capacities. However, this reasoning is restricted to acyclic graphs. Also, [24] provides no algorithm for computing flow footprints.

In a sense, our work strikes a balance between the two prior meta theories by guaranteeing unique flows without sacrificing expressivity and, at the same time, enabling better proof automation. That said, we believe that the framework proposed in [24] remains of independent interest, in particular if the application does not require unique flows (i.e., does not impose lower bounds on flows that may trivially hold in the presence of vanishing flows). Cancellativity allows one to aggregate inflows and outflows to unary functions, which can lead to smaller flow footprints (i.e., more local proofs).

The benchmark suite for our evaluation is obtained from `plankton` [26,27], a tool for verifying concurrent search structures using keyset flows. When the program mutates the symbolic heap, `plankton` creates a flow graph for the mutated nodes plus all nodes with a distance of k or less from those nodes. This flow graph is considered to be the footprint and contextual equivalence is checked. The check is basically the same as for NAIVE. However, the paper does not present the meta theory for the underlying notion of flow graphs, nor does it provide any justification for the correctness of the implemented algorithms used to reason about flow graphs.

Flow graphs form a separation algebra. Hence, the developed theory can be used in combination with any existing separation logic that is parametric in the underlying separation algebra such as [5, 7, 18, 27, 41, 44]. Identifying footprints of updates relates to the frame inference problem in separation logic, which has been studied extensively [4, 6, 15, 25, 35, 36, 42]. However, existing work focuses on frame inference for assertions that are expressed in terms of inductive predicates. These techniques are not well-suited for reasoning about programs manipulating general graphs, including overlayed structures, which are often used in practice and easily expressed using flows. A common approach to reason about general heap graphs in separation logic is to use iterated separating conjunction [14, 39, 44, 47] to abstract the heap by a *pure* graph that does not depend on the program state. Though, the verification of specifications that rely on inductive properties of the pure graph then resorts back to classical first-order reasoning and is difficult to automate. An exception is [45] which uses SMT solvers to frame binary reachability relations in graphs that are described by iterated separating conjunctions. However, the technique is restricted to such reachability properties only.

Unbounded footprints have been encountered early on when computing the post image for recursive predicates [8]. This has spawned interest in separation logic fragments for which the reasoning can be efficiently automated [2,3,9,17,20,35,38]. A limitation that underlies all these works is an assumption of tree-regularity of the heap, in one way or another, which flows have been designed to overcome. In cases where the program (or ghost code) traverses the unbounded footprint (before or after the update), recent works [24, 27] have found a way to reduce the reasoning to bounded footprint chunks.

The definition of a flow closely resembles the classical formulation of a forward data flow analysis. The fact that the least fixed point of the flow equation for distributive edge functions can be characterized as a join over all paths in the flow graph mirrors dual results for greatest fixed points in data flow analysis [19,21]. In a similar vein, the notion of contextual equivalence of flow graphs relates to contextual program equivalence and fully abstract models in denotational semantics [16,30,37]. In fact, Bekić's Lemma [1], which we use in the proofs of Theorem 1 and lemma 6, was originally motivated by the study of such models. Flow graphs can serve as abstractions of programs (rather than just program states). We therefore believe that our results could also be of interest for developing incremental and compositional data flow analysis frameworks.

Data Availability Statement

The `krill` artifact and dataset generated and/or analysed in the present paper are available in the Zenodo repository [28], `https://zenodo.org/record/7566204`.

Acknowledgments

This work is funded in part by NSF grant 1815633. The first author was supported by the DFG project *EDS@SYN: Effective Denotational Semantics for Synthesis*. The third author is supported by a Junior Fellowship from the Simons Foundation (855328, SW).

References

1. Bekić, H.: Definable operation in general algebras, and the theory of automata and flowcharts. In: Programming Languages and Their Definition. Lecture Notes in Computer Science, vol. 177, pp. 30–55. Springer (1984)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: FSTTCS. Lecture Notes in Computer Science, vol. 3328, pp. 97–109. Springer (2004)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer (2005)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: APLAS. Lecture Notes in Computer Science, vol. 3780, pp. 52–68. Springer (2005)
5. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS. pp. 366–378. IEEE (2007)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. pp. 289–300. ACM (2009)
7. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300. ACM (2013)
8. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS. Lecture Notes in Computer Science, vol. 3920, pp. 287–302. Springer (2006)
9. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: SPEN: A solver for separation logic. In: NFM. Lecture Notes in Computer Science, vol. 10227, pp. 302–309 (2017)
10. Feldman, Y.M.Y., Enea, C., Morrison, A., Rinetzky, N., Shoham, S.: Order out of chaos: Proving linearizability using local views. In: DISC. LIPIcs, vol. 121, pp. 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
11. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. Lecture Notes in Computer Science, vol. 2180, pp. 300–314. Springer (2001)
12. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. Lecture Notes in Computer Science, vol. 3974, pp. 3–16. Springer (2005)
13. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
14. Hobor, A., Villard, J.: The ramifications of sharing in data structures. In: POPL. pp. 523–536. ACM (2013)
15. Holík, L., Peringer, P., Rogalewicz, A., Soková, V., Vojnar, T., Zuleger, F.: Low-level bi-abduction. In: ECOOP. LIPIcs, vol. 222, pp. 19:1–19:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
16. Hyland, J.M.E., Ong, C.L.: On full abstraction for PCF: i, ii, and III. Inf. Comput. **163**(2), 285–408 (2000)

17. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: ATVA. Lecture Notes in Computer Science, vol. 8837, pp. 201–218. Springer (2014)
18. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018)
19. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica **7**, 305–317 (1977)
20. Katelaan, J., Zuleger, F.: Beyond symbolic heaps: Deciding separation logic with inductive definitions. In: LPAR. EPiC Series in Computing, vol. 73, pp. 390–408. EasyChair (2020)
21. Kildall, G.A.: A unified approach to global program optimization. In: POPL. pp. 194–206. ACM Press (1973)
22. Krishna, S., Patel, N., Shasha, D.E., Wies, T.: Verifying concurrent search structure templates. In: PLDI. pp. 181–196. ACM (2020)
23. Krishna, S., Shasha, D.E., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. Proc. ACM Program. Lang. **2**(POPL), 37:1–37:31 (2018)
24. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 308–335. Springer (2020)
25. Le, Q.L., Sun, J., Qin, S.: Frame inference for inductive entailment proofs in separation logic. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 41–60. Springer (2018)
26. Meyer, R., Wies, T., Wolff, S.: Artifact for "A Concurrent Program Logic with a Future and History" (Sep 2022). `https://doi.org/10.5281/zenodo.7080459`
27. Meyer, R., Wies, T., Wolff, S.: A concurrent program logic with a future and history. Proc. ACM Program. Lang. **6**(OOPSLA) (2022)
28. Meyer, R., Wies, T., Wolff, S.: Artifact for "Make flows small again: revisiting the flow framework" (Jan 2023). `https://doi.org/10.5281/zenodo.7566204`
29. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. pp. 73–82. ACM (2002)
30. Milner, R.: Fully abstract models of typed *lambda*-calculi. Theor. Comput. Sci. **4**(1), 1–22 (1977)
31. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
32. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001)
33. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC. pp. 85–94. ACM (2010)
34. Patel, N., Krishna, S., Shasha, D.E., Wies, T.: Verifying concurrent multicopy search structures. Proc. ACM Program. Lang. **5**(OOPSLA), 1–32 (2021)
35. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 773–789. Springer (2013)
36. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 711–728. Springer (2014)
37. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977)
38. Qiu, X., Wang, Y.: A decidable logic for tree data-structures with measurements. In: VMCAI. Lecture Notes in Computer Science, vol. 11388, pp. 318–341. Springer (2019)
39. Raad, A., Hobor, A., Villard, J., Gardner, P.: Verifying concurrent graph algorithms. In: APLAS. Lecture Notes in Computer Science, vol. 10017, pp. 314–334 (2016)
40. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)

41. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Tada: A logic for time and data abstraction. In: ECOOP. Lecture Notes in Computer Science, vol. 8586, pp. 207–231. Springer (2014)
42. Rowe, R.N.S., Brotherston, J.: Automatic cyclic termination proofs for recursive procedures in separation logic. In: CPP. pp. 53–65. ACM (2017)
43. Scott, D.: Outline of a mathematical theory of computation. Tech. Rep. PRG02, Oxford University Computing Laboratory (1970)
44. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI. pp. 77–87. ACM (2015)
45. Ter-Gabrielyan, A., Summers, A.J., Müller, P.: Modular verification of heap reachability properties in separation logic. Proc. ACM Program. Lang. **3**(OOPSLA), 121:1–121:28 (2019)
46. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI. pp. 125–135. ACM (2008)
47. Yang, H.: An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In: Proceedings of the SPACE Workshop (2001)