INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Verification of the Flow Framework from "Local Reasoning for Global Graph Properties"

Bernhard Pöttinger

Supervisor: Prof. Tobias Nipkow, PhD
Advisors: Maximilian P. L. Haslbeck
Siddharth Krishna, PhD
Submission Date: 14.09.2020

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 14.09.2020

.....
(Bernhard Pöttinger)

Acknowledgements

I want to thank Prof. Nipkow for the opportunity to attend his outstanding lectures and to do my Master's thesis at his chair. Furthermore, I want to thank my advisor Max Haslbeck for coming up with the interesting topic and for greatly supporting me doing my work. Finally, I want to thank Siddharth Krishna – who became my second advisor in the course of this thesis – for answering and discussing all my questions on his papers in detail. It was a pleasure to learn from and to work with each of you.

Abstract

The Flow Framework [Krishna et al., 2020] introduces a separation algebra that allows handling potentially unbounded side-effects of modifications to global graph properties in a modular fashion. In this thesis we formalize the Flow Framework as presented in "Local Reasoning for Global Graph Properties" [Krishna et al., 2020]. Subsequently, we demonstrate the application of the Flow Framework by verifying a graph algorithm. Finally, we present a significantly refined proof for a central theorem of the framework.

Contents

1. Introduction

1.1. Motivation

Modularity is one of the most important and powerful concepts in computer science: break a problem into smaller problems, solve these smaller problems (or find existing solutions), and compose these building blocks into a solution of the original problem. This concept enabled the construction of systems that no human can grasp in their entirety anymore. To a certain extent it is a surprise that these systems work at all. The essential property of modularity is that each component is constructed independently from other components. This approach tremendously reduces the number of interactions that have to be considered during construction of systems.

An instance that definitely demonstrates the power of modularity is software verification. Until around 2000 reasoning about mutable heap data structures involved significant efforts to cope with aliasing of pointers. Then, [Reynolds, 2002] discovered the separating conjunction – a logical connective that allows programmers to express their intuition about pointers and the heap formally. The key property of the separating conjunction is to prevent overlapping heap regions and therefore to prevent pointer aliasing. The absence of pointer aliasing enables modular reasoning about the heap as fatal side-effects invalidating logical descriptions of the heap are eliminated. Modular reasoning about the heap enables software verification to scale beyond trivial examples and to be deployed in industry to check code-bases of millions of lines of code [Pym et al., 2018, O'Hearn, 2019].

However, despite enabling the efficient description of heap data structures the separating conjunction also imposes significant constraints on these descriptions. Many data structures, for example graphs, exhibit a degree of inherent sharing between parts of the data structure that is difficult to capture using the separating conjunction [Hobor and Villard, 2013]. Furthermore, global properties like reachability in graphs are usually not compatible with the modular reasoning introduced by the separating conjunction: modifications to separated heap portions of data structures can invalidate global properties that are valid for the separated parts of the data structure [Hobor and Villard, 2013, Krishna, 2019]. Describing such data structures requires significant efforts to work around the constraints imposed by the separating conjunction, see for example [Hobor and Villard, 2013].

In this thesis we verify and apply a new approach called the *Flow Framework* [Krishna et al., 2020, Krishna, 2019] that tries to improve this situation by introducing a new formalism to describe global properties of data structures in a local fashion. In contrast to the usual inductively defined predicates that only allow very limited and laborious possibilities to encode ambiguous representations of heap data structures the new formalism is able to adapt dynamically to arbitrary representations of such data structures: it represents data structures as unstructured heap portions and allows us to "materialize" and modify excerpts of data structures when and as needed.

1.2. Structure

Chapter 2 presents a short motivation of the Flow Framework starting from the basics of Hoare logic and Separation Logic. Chapter 3 introduces some preliminary development of auxiliary theories that will play important roles in the formalization of the Flow Framework. Chapter 4 then describes our formalization of the Flow Framework. Chapter 5 verifies the correctness of a graph update algorithm using the Flow Framework. Chapter 6 closes this thesis by giving a refined proof of theorem 3 from [Krishna et al., 2020].

1.3. Contributions

This thesis verifies the verification framework called *Flow Framework* presented in [Krishna et al., 2020] and [Krishna, 2019]. More concretely the contributions of this thesis are:

1. Formalization of the Flow Framework in Isabelle/HOL.
2. Several small corrections to [Krishna, 2019, Krishna et al., 2020].
3. End-to-end formalization and verification of an algorithm using our implementation of the Flow Framework. Additionally, termination of the algorithm is proven.
4. Counter-example and fix for theorem 3 from [Krishna et al., 2020].
5. Significant refinement of the proof of theorem 3 from [Krishna et al., 2020].

1.4. Notation

Sometimes we denote function application using $x \triangleright f \equiv f\ x$ and function composition using $f_1 \triangleright f_2 \equiv f_2 \circ f_1$. We will use `combine` to combine two functions into a single one depending on disjoint sets N_1 and N_2. Outside N_1 and N_2 the result function assumes a default value x_0:

$$\texttt{combine}\ N_1\ N_2\ x_0\ f_1\ f_2 = \lambda x.\ \begin{cases} f_1\ x & \text{for } x \in N_1 \\ f_2\ x & \text{for } x \in N_2 \\ x_0 & \text{otherwise} \end{cases}$$

To restrict functions to given domains and to set a default value outside these domains we introduce

$$\texttt{restrict}\ N\ x_0\ f = \lambda x.\ \begin{cases} f\ x & \text{for } x \in N \\ x_0 & \text{otherwise} \end{cases}$$

We will use a function δ:

$$\delta_{x=x'} \coloneqq \delta\ x\ x' \coloneqq \begin{cases} \mathsf{id} & \text{for } x = x' \\ \lambda\_.\ 0 & \text{otherwise} \end{cases}$$

We will represent partial functions as $\{x \mapsto a, y \mapsto b, \ldots\}$.

2. Basics

2.1. Hoare Logic

This section provides a short introduction to Separation Logic [Reynolds, 2002].

Separation Logic is an extension of Hoare logic [Hoare, 1969]. Therefore, we start our introduction with a short detour to Hoare logic based on [Nipkow and Klein, 2014]. Hoare logic enables us to prove program specifications by deriving the specifications using a given set of inference rules. In Hoare logic specifications for programs c are stated as triples

$$\{P\}\ c\ \{Q\}$$

where P is a logical assertion called *pre-condition* and Q is a logical assertion called *post-condition*. The semantics of such a specification is that if an execution of a program c is started in state s and s satisfies pre-condition P and if there is a state s' resulting from the execution of c then s' satisfies post-condition Q. The stated semantics ignores termination, i.e. only proves partial correctness. For total correctness termination must be additionally proven. In order to derive proofs for program specifications we are provided with a set of inference rules that enable us to compositionally prove program specifications by splitting proofs into smaller proofs according to the the program structure and recombining these "small" proofs into proofs for the entire specification.

Consider the toy programming language IMP generated by the following grammar:

$$c ::= \texttt{SKIP} \mid x{:=}e \mid c_1; c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c$$

IMP provides the no-op command `SKIP`, assignments to variables, sequencing of commands, case distinctions and while loops. b is supposed to represent boolean expressions and e is supposed to represent arithmetic expressions. The corresponding inference rules from Hoare logic are

$$\frac{}{\{P\}\ \texttt{SKIP}\ \{P\}}\ \text{SKIP} \qquad \frac{}{\{P[x \mapsto e]\}\ x{:=}e\ \{P\}}\ \text{ASSIGN}$$

$$\frac{\{P_1\}\ c_1\ \{P_2\} \qquad \{P_2\}\ c_2\ \{P_3\}}{\{P_1\}\ c_1; c_2\ \{P_3\}}\ \text{SEQ} \qquad \frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}\ \text{ITE}$$

$$\frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{P \wedge \neg b\}}\ \text{WHILE} \qquad \frac{P' \longrightarrow P \qquad \{P'\}\ c\ \{Q'\} \qquad Q \longrightarrow Q'}{\{P'\}\ c\ \{Q'\}}\ \text{CONS}$$

Rule SKIP states that after the execution of `SKIP` the same assertion as before still holds. Rule ASSIGN describes the effect of assignments. This rule exhibits the backwards reasoning style that is inherent in Hoare logic, i.e. we derive a pre-condition starting from the post-condition and not the other (more intuitive) way round. Rule SEQ proves the correctness of command sequences. This rule exhibits the compositional nature of Hoare logic: the proof for $c_1; c_2$ is split into two proofs, one for c_1 and another one for c_2, and then both proofs are combined to a proof for $c_1; c_2$ by requiring that the post-condition of c_1 equals the pre-condition of c_2. Rule ITE introduces a case-distinction in our proof: one case if guard b holds, another case if guard b does not hold.

In both cases this additional information, i.e. b and $\neg b$, respectively, is added to the respective case such that this information can be exploited in the corresponding proofs. Both cases have to ensure the same post-condition. Rule WHILE enables us to reason about loops. The difficulty that this rules has to incorporate is that we do not know for how many iterations the loop will actually execute (if any at all). The central ingredient to tackle this difficulty is the *loop invariant*. Rule WHILE expects loop invariants to hold before the loop is executed and that each iteration reestablishes the loop invariant. Therefore, independently from the number of actual iterations during runtime we know that the state after the execution again is suited for another iteration. When the loop eventually terminates we know that the final state also satisfies the loop invariant, hence the loop invariant is also the post-condition. The only additional information that is available for the proof of the loop body and the post-condition, respectively, is if loop condition b holds in the current state or not: the pre-condition of the loop body can assume that loop condition b (still) holds, the post-condition can assume that loop-condition b does not hold anymore. These manifold roles of loop invariants renders finding loop invariants the central challenge of Hoare logic while all other rules can be applied in a mechanical fashion. Finally, rule CONS enables us to weaken and strengthen pre-condition and post-condition, respectively, in order to omit or infer information from assertions.

As a short example we derive a proof for a program computing $x \operatorname{idiv} y$, i.e. the integer k such that $0 \leq x - ky < y$. Assuming input $x \geq 0$ we compute k using the program fragment

$$k{:=}0; \texttt{while } x \geq y \texttt{ do } (k{:=}k+1; x{:=}x-y)$$

We apply the loop invariant $I \equiv x \geq 0 \wedge x_0 = x + ky$. Hoare-style proofs usually do not use derivation trees but embed assertions into the program text. As our program fragment modifies variable x we introduced a logical variable x_0 in our invariant that represents the initial value of x for the entire scope of the program and is used for verification purposes only:

$$\begin{array}{l}
\{x \geq 0 \wedge x_0 = x\} \\
k{:=}0; \\
\{x \geq 0 \wedge x_0 = x \wedge k = 0\} \\
\{x \geq 0 \wedge x_0 = x + ky\} \\
\texttt{while } x \geq y \texttt{ do } (\\
\quad \{x \geq y \wedge x \geq 0 \wedge x_0 = x + ky\} \\
\quad \{x - y \geq 0 \wedge x_0 = (x-y) + (k+1)y\} \\
\quad k{:=}k+1; \\
\quad \{x - y \geq 0 \wedge x_0 = (x-y) + ky\} \\
\quad x{:=}x-y \\
\quad \{x \geq 0 \wedge x_0 = x + ky\} \\
) \\
\{\neg(x \geq y) \wedge x \geq 0 \wedge x_0 = x + ky\} \\
\{0 \leq x_0 - ky < y\}
\end{array}$$

Hoare logic works very well for programs manipulating primitive data types like integers or strings. However, more complex programs that involve mutable heap data structures impose significant complexity in Hoare logic. [O'Hearn, 2019]

To understand this statement we study the simple example of concatenating two finite singly-linked lists in Figure 2.1 using Isabelle's existing Hoare logic formalization that includes facilities

```
lemma "VARS next x y u
{x ≠ Null ∧ List next x Xs ∧ List next y Ys ∧ set Xs ∩ set Ys = {}}
  u := x;
  WHILE u^.next ≠ Null
  INV {∃us vs. x ≠ Null ∧ u ≠ Null ∧ u ∈ set (map Ref Xs) ∧ Xs = us @ vs ∧
          Path next x us u ∧ Path next u vs Null ∧ List next y Ys ∧
          set Xs ∩ set Ys = {} ∧ distinct Xs}
  DO
    u := u^.next
  OD;
  u^.next := y
{List next x (Xs @ Ys)}
"
```

Figure 2.1.: Example: Append lists in Hoare logic

```
lemma "VARS H x y u
{(list x Ps ** list y Qs) H ∧ x ≠ 0}
  u := x;
  WHILE the (H u) ≠ 0
  INV {∃us vs. (lseg x us u ** lseg u vs 0 ** list y Qs) H
               ∧ Ps = us @ vs ∧ u ≠ 0 ∧ x ≠ 0}
  DO
    u := the (H u)
  OD;
  H := H(u ↦ y)
{(list x (Ps @ Qs)) H}
"
```

Figure 2.2.: Example: Append lists in Separation Logic

to reason about heaps. In the example in Figure 2.1 the representation of heaps needs some explanation: for each field of a data structure we have to introduce a dedicated "heap". Our singly-linked lists only consist of a single field `next`. Therefore, we introduce a heap `next` :: '$a \Rightarrow$ 'a (see first line, we omit the framework's `ref` type wrapper), where type 'a represents the type of memory locations. To access a field `field` at memory location x we have to use the notation x^.`field`. Lists are represented by a predicate `List` $h\ x\ xs \equiv$ `Path` $h\ x\ xs$ `Null`. `Path` $h\ x\ xs\ y$ describes that in heap h we have a list segment starting in node x and terminating in node y by following the pointers in h. The terminal node y is not considered to be part of the list segment. The logic-level list xs contains the sequence of nodes part of the heap list segment (excluding the terminal node y).

Our program fragment in Figure 2.1 expects two lists starting in nodes x and y, respectively (in the following we simply identify lists by their head nodes). x is furthermore expected to be non-empty. The program fragment traverses list x until reaching x's last node and connects x's last node to y. The result is the concatenated list starting in node x.

The key observation in this example is that the node sequences Xs and Ys of x and y, respectively, are required to be disjoint, i.e. $\mathsf{set}\ Xs \cap \mathsf{set}\ Ys = \emptyset$. To understand why we need this assumption consider the following example where two lists x and y that share a suffix list segment $x_1, \ldots, x_3$ are appended:

Obviously, append constructs a cyclic list. As our lists are supposed to be finite they in particular are not supposed to contain cycles. To avoid this and many similar faulty scenarios we have to explicitly ensure that both lists do not share nodes, i.e. are disjoint.

Unfortunately, sharing is an even more significant problem than we just saw: consider a third list z that does not participate in the above program fragment but exists in the same heap as x and y. Inspecting our program provides no hints that z is affected by our program as z is not mentioned within the program. However, if z shares nodes with x then appending y to x obviously also appends y to z:

This behavior is even more problematic than the one before for two reasons: first, the effect is not apparent from the program text, and second, to prevent this behavior we have to ensure that all $\mathcal{O}(n^2)$ pairs of instances of data structures within the program do not share memory. The only way to prevent sharing in Hoare logic is to explicitly state that the heap locations of each pair of instances are disjoint. Obviously, the $\mathcal{O}(n^2)$ issue prevents verification of heap programs using Hoare logic to scale to non-trivial dimensions, and also runs directly counter to best-practices of software engineering, in particular modularity: we only want to consider details that are relevant for a particular problem and we want to ignore details that are irrelevant to the problem. Instances of data structures not participating in an operation obviously should be irrelevant to the verification of the operation.

2.2. Separation Logic

To solve Hoare logic's sharing issue Separation Logic enhances the assertion logic of Hoare logic: while Hoare logic employs plain predicate logic to express assertions Separation Logic uses an instance of the "logic of bunched implications", short BI. BI [O'Hearn and Pym, 1999] enables reasoning about finite resources, and heaps can be considered as finite resources [Reynolds, 2002]. Each memory location is a resource and there is exactly one instance of each memory location. The key insight of BI is that reasoning about finite resources implies the need for some concept of availability and management of resources: only available resources can be used, and if too many resources are requested then there is a problem. Multiple requests of resources can be stated using BI's multiplicative conjunction $\varphi * \psi$. Given a structure r (which has to be an element of a partial commutative monoid $(R, \cdot, 0)$) representing the available resources the semantics of the multiplicative conjunction is:

$$r \models \varphi * \psi \quad \text{iff. there exist } s, t \in R \text{ such that } r = s \cdot t \text{ and } (s \models \varphi) \text{ and } (t \models \psi)$$

i.e. structure r can be split into two complementary "sub-structures" s and t such that s is able to satisfy the resource requests of φ and t is able to satisfy the resource requests of ψ. [Pym et al., 2018]

In the case of Separation Logic's heap instance (using partial finite maps $\mathsf{Val} \rightharpoonup_{\mathsf{fin}} \mathsf{Val}$ with $m_1 \cdot m_2 := \texttt{if } \mathsf{dom}\, m_1 \cap \mathsf{dom}\, m_2 = \emptyset \texttt{ then } (\lambda x.\ \texttt{if } x \in \mathsf{dom}\, m_1 \texttt{ then } m_1\ x \texttt{ else } m_2\ x) \texttt{ else } \bot$

and $0 := \emptyset$ as structures) of BI availability means that each of the memory locations can be used by clients only if they have exclusive access to the memory location. Availability (or "ownership") is indicated by a token, called points-to assertion: $x \mapsto y$. This token asserts that there is an owned and therefore accessible memory location x, and moreover that the content of this memory location is y. In Separation Logic the multiplicative conjunction is called "separating conjunction". For example $\varphi \equiv x \mapsto x_0 * y \mapsto y_0$ represents the ownership of two memory locations x and y with contents x_0 and y_0, respectively. The exclusive access model of Separation Logic in particular implies that formula $x \mapsto x_0 * x \mapsto x_0$ is unsatisfiable.

In order to access the heap Separation Logic introduces dedicated commands to the programming language:

$$c ::= \ldots \mid [e_1] := e_2 \mid x := [e]$$

The corresponding inference rules that implement the ownership-based[1] heap model are:

$$\frac{}{\{x \mapsto e\}\ [x] := e'\ \{x \mapsto e'\}}\ \text{WRITE} \qquad \frac{}{\{e \mapsto e'\}\ x := [e]\ \{e \mapsto e' \wedge x = e'\}}\ \text{READ}$$

These rules explicitly require the ownership of the accessed heap locations. If the accessed heap location is not available then the derivation of a proof fails. For simplicity of presentation we omitted side-conditions of these rules.

There are two aspects how Separation Logic solves Hoare logic's heap issue: first, the ownership-based access model allows program fragments to only alter portions of the heap that they are explicitly allowed to alter; and second, the separating conjunction enables us to keep instances of data structures separated, i.e. there can not be sharing between two instances described by assertions φ and ψ if $\varphi * \psi$ holds. Therefore: if we provide heap portions to a program fragment and these heap portions are separated from the remaining heap then we can be sure that none of the remaining parts of the heap are affected by the execution of the program fragment because the ownership-based heap model restricts the program fragment's heap accesses the provided heap portions and the remaining heap portions do not overlap with the provided ones. This reasoning is captured by Separation Logic's frame rule:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * F\}\ c\ \{Q * F\}}\ \text{FRAME}$$

The narrative of the FRAME rule can be given as follows: the FRAME rule restricts the heap portions provided to command c to P by holding back the so-called *frame* F from c. c then executes and provides back some usually modified heap portion Q. As c is subject to the ownership-based heap model we can be sure that F was not affected by the execution of c, therefore after the execution of c we know that F still holds and we can recompose Q and F to obtain the post-condition.

From the perspective of command c not having to consider irrelevant parts of the heap coins the term *local reasoning*. In contrast we use to term *global reasoning* for contexts where we can not restrict our attention to the relevant parts of the heap. Therefore, this restriction actually provides an advantage to c by being able to be specified and verified independently from irrelevant portions of the heap. The portions of the heap that are required by a program fragment to work properly are called *footprint*. Of course, the ownership-based heap model now requires us to explicitly include the footprint of program fragments in program specifications.

As a short prospect, a not too far-fetched (but hard to proof) generalization of the FRAME

[1] we could also call the model "permission-based"

rule provides us Concurrent Separation Logic [O'Hearn, 2007, Brookes, 2007]:

$$\frac{\{P_1\}\ c_1\ \{Q_2\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 * P_2\}\ c_1 \,||\, c_2\ \{Q_1 * Q_2\}}\ \text{PAR}$$

Instead of keeping a frame F unused for the duration of the execution of command c, we can provide F to a second command which executes concurrently to the first one. The separating conjunction ensures that there are no data races between the heap portions described by P and F. Of course, this concurrency model is very strict as it prevents shared memory communication between c_1 and c_2. However, mutexes including ownership transfer between threads can be modelled quite naturally using the ownership narrative (see [O'Hearn, 2007]). There are also enhancements that enable atomic operations, e.g. [Vafeiadis and Parkinson, 2007]. The development of Iris [Jung et al., 2018] probably can be considered the state-of-the-art of Separation Logic.

We revisit our example on appending two lists using Separation Logic (see Figure 2.2). In order to be able to compare this version of the example with the previous one we apply Isabelle's existing formalization of separation algebras [Klein et al., 2012] that is supposed to be embedded into the same Hoare logic framework that we already used in the previous example. Therefore, we again have to create an explicit heap H and Separation Logic assertions have to be provided explicitly with H. To access heap H at memory location x we have to use $H\ x$. As H is only a partial mapping we have to unwrap the content of the memory location using `the` $(H\ x)$. To update a memory location in H we simply update heap H accordingly in Isabelle's map update syntax $H\ (x \mapsto x')$.

Our program fragments in Figure 2.1 and Figure 2.2 only differ in the notation for heap accesses. The program specifications differ to that effect that we no longer have to explicitly state that the appended lists are disjoint. Separation Logic enables us to express this property implicitly using the separating conjunction $**$ (special notation of this particular framework). The more significant effect can not be demonstrated in these simple examples: in contrast to Figure 2.1 additional instances of data structures are guaranteed to be not affected by the program fragment in Figure 2.2. The footprint of the program fragment in Figure 2.2 is lists x and y and nothing more. The ownership-based heap model ensures that no other instances than the stated ones can be accessed and the separating conjunction ensures that there is no sharing between x, y and any other instance.

To summarize the advantages of Separation Logic: Separation Logic enables efficient reasoning involving the heap by enabling modular reasoning about the heap and resources in general. Not only in terms of functional correctness but it also provides general guarantees on memory safety (no accesses to uninitialized memory, no memory leaks) along the way. Furthermore, Concurrent Separation Logic prevents data races. [O'Hearn, 2019, Vafeiadis, 2011]

Of course, also Separation Logic has limitations. One of Separation Logic's significant problems is entailment checking: given two Separation Logic formulas φ and ψ, is each model for φ also a model for ψ, i.e. does $(s, h) \models \varphi$ imply $(s, h) \models \psi$ for all models (s, h)[2]? Entailment checking is undecidable for Separation Logic [Antonopoulos et al., 2014]. However, there are multiple known decidable fragments of Separation Logic that apply different strategies to cope with undecidability: from decidable logics with limited expressiveness to more expressive but incomplete ones [Katelaan et al., 2019] that require human assistance.

Another problem of Separation Logic is that data structures are usually described using inductively defined predicates. A particularly difficult class of data structures to formalize using

[2] A partial function $s :: \mathsf{Var} \rightharpoonup_{\mathsf{fin}} \mathsf{Val}$ representing the stack, a partial function $h :: \mathsf{Val} \rightharpoonup_{\mathsf{fin}} \mathsf{Val}$ representing the heap.

inductively defined predicates are graphs. A reason is that there might be multiple potential traversal patterns through graphs and we can not simply partition all reachable nodes into multiple disjoint heap portions as expected by inductively defined predicates because there might be nodes that are reachable via multiple neighbor nodes. An approach to cope with this scenario is to come up with lemmas that convert between multiple representations of data structures. However, these lemmas are usually expensive to prove. In order to address this "splitting issue" [Hobor and Villard, 2013] introduce some interesting approach to reason about such data structures: they apply weaker logical connectives from separation logic that allow sharing and manage the shared parts using higher-level mathematical structures. However, the applied advanced logical connectives from separation logic that are hard to automate due to decidability issues if the magic wand [Krishna, 2019].

2.3. The Flow Framework

The graph splitting issue is one of the motivations of the primary source of this thesis [Krishna et al., 2020] which tries to approach this and other limitations of Separation Logic. Their idea is to not impose a fixed structure like tree predicates on heap objects but to follow an existing approach that maintains an unstructured set of heap objects using the iterated separating conjunction. These heap objects are arranged into local excerpts of data structures just-in-time during verification. Operations on data structures first obtain the required excerpt, then apply their modifications, and finally dissolve the excerpt such that the involved heap locations return into the unstructured set of heap objects.

Of course, this approach does not allow us anymore to encode (non-trivial) invariants of data structures into logical structures like inductively defined predicates that also structure related heap objects. The key contribution of [Krishna et al., 2020] is a new approach to state invariants in a more flexible way in the context of sets of unstructured heap objects. They introduce so-called flow graphs that exchange pieces of information about heap objects. These pieces of information then are used to state invariants on single heap objects using the exchanged information. Together, all these "heap object-local" invariants are supposed to imply the property that is to be proven.

We introduce and formalize this approach to describe invariants in Chapter 4. Section 5.1 then connects these invariants with the unstructured approach to managing heap objects.

3. Preliminary Development

This chapter presents some more or less self-sustaining theories that emerged from the formalization of the Flow Framework. To simplify the presentation of the Flow Framework's formalization we present these theories on their own in this chapter.

3.1. Alternating List Predicates

Our formalization of the Flow Framework will involve paths $xs = [x_1, \ldots, x_n]$ within graphs h that are composed from two subgraphs h_1 and h_2, i.e. $h = h_1 + h_2$ for some notion of graph composition. In this context we will have to reason about decompositions of xs into path segments xs_i, i.e. consecutive subpaths xs_i of xs that run entirely within h_1 or h_2 and $\mathsf{concat}\,[xs_1, \ldots, xs_k] = xs$. We call a list of path segments a *convoluted path*, i.e. $xss = [xs_1, \ldots, xs_k]$ is a convoluted path.

To model this requirement we first introduce alternating list predicates `alternating` $P\,Q$:

```
fun alternating :: "('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ bool" where
  "alternating _ _ [] = True" |
  "alternating P Q (x # xs) = (P x ∧ alternating Q P xs)"
```

Alternating list predicates allow us to specify for lists $xs = [x_1, \ldots, x_n]$ that $P\,x_i$ holds for all odd indices i and that $Q\,x_i$ holds for all even indices i.

A second variant of `alternating` explicitly states a last element y:

```
fun alternating' where
  "alternating' P Q [] y = P y" |
  "alternating' P Q (x # xs) y = (P x ∧ alternating' Q P xs y)"
```

This variant mirrors the structure of later developments that state some terminal element explicitly and usually depend on this information within proofs of associated lemmas. To avoid manual case distinctions in these proofs we introduce this second variant of alternating list predicates.

To model that multiple consecutive elements within a list satisfy the same predicate P, like for example that multiple consecutive nodes of a path belong to the same subgraph, we simply represent the original list xs as a list of lists $xss = [xs_1, \ldots, xs_k]$ with $xs = \mathsf{concat}\,xss$ and lift single element predicates P to list predicates $P' \equiv \lambda xs.\ \forall x \in \mathsf{set}\,xs.\ P\ x$.

Now that we are able to describe alternating lists we introduce some induction rules that significantly simplify inductive reasoning over alternating lists. The induction scheme that we will encounter multiple times is the following one: given a convoluted path xss with alternating predicates P and Q, we have to show some proposition $R\ xss$. We want to do steps of one path segment at a time. In order to do so for a current path segment xs_i we usually not only need the current path segment's predicate $P\ xs_i$ but also the successor path segment's predicate $Q\ xs_{i+1}$ to reason about the connection between both path segments. Of course we have to consider the succession "first Q, then P" analogously. The following induction rule captures this reasoning for `alternating` (there is also an analogous variant for `alternating'` that incorporates the terminal element):

$$\dfrac{\texttt{R b bb a aa []} \qquad \bigwedge \texttt{x. } \dfrac{\texttt{alternating P Q xs} \qquad \texttt{R a aa b bb []}}{\texttt{alternating P Q [x]} \wedge \texttt{P x}} \qquad \bigwedge \texttt{x. } \dfrac{\texttt{alternating Q P [x]} \wedge \texttt{Q x}}{\texttt{R b bb a aa [x]}}}{\dfrac{\bigwedge \texttt{x y zs. } \dfrac{\texttt{alternating P Q (x · y · zs)} \wedge \texttt{P x} \wedge \texttt{Q y} \wedge \texttt{R b bb a aa (y · zs)}}{\texttt{R a aa b bb (x · y · zs)}}}{\dfrac{\bigwedge \texttt{x y zs. } \dfrac{\texttt{alternating Q P (x · y · zs)} \wedge \texttt{Q x} \wedge \texttt{P y} \wedge \texttt{R a aa b bb (y · zs)}}{\texttt{R b bb a aa (x · y · zs)}}}{\texttt{R a aa b bb xs}}}$$

Basically this rule is an improved version of `induct_list012` that explicitly presents us all the cases that we would induce manually inside proofs if we used `induct_list012` itself. This rule prevents us from repeating these case distinctions all over again and significantly simplifies reasoning about alternating lists. The parameterization `a`, `aa`, `b`, `bb` of proposition `R` enables us to include two *alternating variables* in `R`. For example, we will have to refer to some property of the head node of a convoluted path *xss* that depends on the subgraph that the head node is part of. Our proof will start with the full convoluted path *xss* and it is known that its head node is in e.g. subgraph h_1. However, the next proof step during the induction will consider the convoluted path $\mathsf{tl}\, xss$ and that convoluted path starts in the other subgraph h_2. Therefore, `R` must be able to talk about a variable that alternatingly assumes values h_1 and h_2. Parameterization `a` and `b` constitute the first alternating variable, and `aa` and `bb` constitute a second alternating variable, i.e. `a` and `b`, and `aa` and `bb` are swapped with each inductive step.

Sometimes we will have to refer to the value of an alternating variable that is active when we encounter the last element of a list. Obviously, we can not know which one of the two alternating variables corresponds to the end of the alternating list as the length of the list can be even or odd. To state such an unknown variable we introduce the following function:

```
fun alt :: "'a ⇒ 'a ⇒ 'b list ⇒ 'a" where
  "alt _ y [] = y" |
  "alt x y (_ # xs) = alt y x xs"
```

If `alt a b xs` is included in an alternating proposition `R a aa b bb` then `a` and `b` in `alt a b xs` swap in parallel to the parameterization of `R` during induction and therefore `alt a b xs` always refers to the same variable. Of course we still do not know to which one of the two alternating variables the term will finally evaluate to. In that case we will simply have to continue our proof with the term `alt a b xs`. In some cases we can infer the value of `alt a b xs` from additional information available in some proof context. If we are able to prove some result independently from the concrete value of `alt a b xs` then rule

```
lemma alt_cases:
  assumes "alt x y zs = x ⟹ P x" "alt x y zs = y ⟹ P y"
  shows "P (alt x y zs)"
```

is useful. Some further simplification rules that will become handy in our formalization are

```
lemma alt_alt_to_alt: "alt (alt P Q xs) (alt Q P xs) ys = alt Q P (xs @ ys)"

lemma alt_P_P_hom: "alt (P a) (P b) xs = P (alt a b xs)"
```

To split a list into a convoluted list according to two sets A and B we have

```
lemma split_segments:
  assumes "xs ≠ []" "set xs ⊆ A ∪ B" "hd xs ∈ A"
  shows "∃xss. concat xss = xs ∧ xss ≠ [] ∧ (∀xs ∈ set xss. xs ≠ []) ∧
    alternating (segment A) (segment B) xss"
```

Here, `segment X` $\equiv \lambda$`xs. set xs` $\subseteq$ `X` $\wedge$ `xs` $\neq []$. We do not require that A and B are disjoint. The head element of xs is required to belong to A as `alternating` P Q xs enforces $\mathsf{hd}\, xs$ to satisfy P and our P here requires $\mathsf{hd}\, xs \neq []$. Assumption $xs \neq []$ merely ensures that the resulting list xss is non-empty. We also have a sibling lemma `split_segments`$'$ that does not impose this restriction.

To extend existing convoluted lists with additional elements we provide a relatively specific lemma:

```
lemma split_segments_extend:
  assumes some: "alternating (segment h1) (segment h2) xss" "xss ≠ []"
    "ys = concat xss @ zs" "zs ≠ []" "h1 ∩ h2 = {}" "set zs ⊆ h1 ∪ h2"
    "segment h1 zs ⟷ ¬ segment h2 zs" "segment h1 zs2 ⟷ ¬ segment h2 zs2"
    "segment h1 zs ⟷ segment h2 zs2"
  shows "∃ys' yss.
    alternating' (segment h1) (segment h2) (butlast xss @ [last xss @ ys'] @ yss) zs2 ∧
    ys = concat (butlast xss @ [last xss @ ys'] @ yss) ∧ (∀xs ∈ set xss. xs ≠ []) ∧
    last xss @ ys' ≠ [] ∧ (∀xs ∈ set yss. xs ≠ []) ∧ (ys' @ concat yss = zs) ∧
    (ys' = [] ∨ yss = [])"
```

`split_segments_extend` is tailored to our needs later: we have to extend an existing convoluted list xss with additional elements zs. We know that those elements entirely belong to either A or B. If zs belongs to the same set as $\mathsf{last}\, xss$ then we append zs to $\mathsf{last}\, xss$ by providing $ys' \coloneqq zs$ and $yss' \coloneqq []$. Otherwise we append an additional path segment to xss by providing $ys' \coloneqq []$ and $yss' \coloneqq [zs]$. Our use case additionally requires us to include the successor path segment zs_2 of zs into our considerations. Our use case provides us with enough information on zs and zs_2 to directly construct the expected instance of `alternating`$'$ that terminates in zs_2. The specificity of this lemma is required to obtain the proposition $ys = \emptyset \vee yss = \emptyset$ which significantly simplifies our later use case.

The symmetric cases in the above induction rule are quite inconvenient as usually those symmetric cases can be proven together. To simplify the above induction rule for such cases and to avoid the significant amount of code duplication caused by symmetric proof cases we additionally introduce a symmetric variant of the above induction rule:

$$\frac{\begin{array}{c}\texttt{alternating (P a b) (P b a) xs}\\ \hline \texttt{R a b xs}\end{array}\qquad \bigwedge \texttt{a b. R a b []} \qquad \bigwedge \texttt{a b x.}\ \frac{\texttt{alternating (P a b) (P b a) [x]} \wedge \texttt{P a b x}}{\texttt{R a b [x]}} \qquad \bigwedge \texttt{a b x y zs.}\ \frac{\texttt{alternating (P a b) (P b a) (x · y · zs)} \wedge \texttt{P a b x} \wedge \texttt{P b a y} \wedge \texttt{R b a (y · zs)}}{\texttt{R a b (x · y · zs)}}}{\texttt{R a b xs}}$$

This variant employs only a single predicate P instead of two ones. However, P is now parameterized analogously to `R`. The proof cases now all-quantify over the parameterization of P, i.e. imply both symmetric cases in the above induction rule.

3.2. Endomorphisms

The Flow Framework extensively uses endomorphisms to come up with methods to calculate solutions for or to preserve solutions of implicit equation systems.

We formalize endomorphisms over cancellative commutative monoids as follows:

```
definition endo :: "('m ⇒ 'm :: cancel_comm_monoid_add) ⇒ bool" where
  "endo f ≡ ∀x1 x2. f (x1 + x2) = f x1 + f x2"
```

The set of all endomorphisms is denoted as

```
abbreviation "End ≡ { f . endo f }"
```

The usual properties of endomorphisms apply, e.g. for some $e \in$ `End` it holds that $e\ 0 = 0$, and $e\ x \neq 0 \Longrightarrow x \neq 0$. Of course, most results on endomorphisms that we present in the following are also available lifted to iterated sums.

We will have to consider sub-classes of endomorphisms. To avoid issues regarding the question whether sums of endomorphisms or function compositions are defined we will enforce that these sub-classes are closed under function composition and addition. We introduce a predicate that describes such closed sets of endomorphisms:

```
definition End_closed where
  "End_closed E ≡ E ⊆ End ∧
    (∀f1 f2. f1 ∈ E ⟶ f2 ∈ E ⟶ f1 + f2 ∈ E ∧ f2 o f1 ∈ E)"
```

In addition to cancellative commutative monoids we introduce positive cancellative commutative monoids. Positive monoids correspond to Isabelle's `canonically_ordered_monoid`:

```
class pos_monoid_add = canonically_ordered_monoid_add
class pos_cancel_comm_monoid_add = pos_monoid_add + cancel_comm_monoid_add
```

Using positive cancellative commutative monoids we can prove monotonicity for endomorphisms over these monoids. Our most general lemma in this regard is:

```
lemma pos_endo_mono_closed:
  fixes f1 f2 g1 g2 :: "'m ⇒ 'm :: pos_cancel_comm_monoid_add"
  assumes "f1 ≤ g1" "f2 ≤ g2" "f1 ∈ E" "g1 ∈ E" "End_closed E"
  shows "f1 o f2 ≤ g1 o g2"
```

Sub-classes of endomorphisms that will become relevant later are constant, nilpotent and reduced endomorphisms. We introduce predicates that describe constant, nilpotent and reduced sets of functions that are supposed to be used in conjunction with `End_closed` to describe sets of nilpotent or reduced endomorphisms:

```
abbreviation "const E ≡ ∀e ∈ E. ∃a. e = (λ_. a)"

definition nilpotent :: "('a ⇒ 'a :: zero) set ⇒ bool" where
  "nilpotent E ≡ ∃p > 1. ∀f ∈ E. f^^p = (λ_. 0)"

definition reduced where
  "reduced E ≡ ∀e ∈ E. e o e = 0 ⟶ e = 0"
```

The existentially quantified number `p` in the definition of `nilpotent` is called nilpotency index of `E`. Nilpotent functions are functions that eventually become the zero-function when applied iteratively. We are not sure how useful this definition of nilpotent functions is on its own, however in conjunction with `End_closed` E we obtain the usual definition of nilpotent endomorphisms.

We need a second variant of `reduced` functions: unfortunately, our proof attempt of a theorem of the Flow Framework uncovered a flaw in that theorem regarding the interpretation of `reduced`'s quantification. Instead we were able to finish the proof by using the flawed but stronger interpretation of reduced:

```
definition pointwise_reduced where
  "pointwise_reduced E ≡ ∀e ∈ E. ∀x. e x ≠ 0 ⟶ e (e x) ≠ 0"
```

```
lemma pointwise_reduced_reduced:
  "pointwise_reduced E ⟹ reduced E"
```

Fortunately, the stronger formulation still covers some of the examples (i.e. the one that we checked so far) from [Krishna et al., 2020] and therefore, our proof of that theorem remains useful.

3.3. Function Chains

Our proofs will require us to consider concatenations of edge functions along paths (we call such concatenations *path functions*), i.e. given a function e that provides edge functions $e\ x\ y$ for edges (x, y) we will have to consider path functions $e\ x_1\ x_2 \triangleright e\ x_2\ x_3 \triangleright \ldots \triangleright e\ x_{n-1}\ x_n$ for paths $xs = [x_1, \ldots, x_n]$. To abstract this kind of function composition we introduce *function chains*:

```
fun chain :: "('a ⇒ 'a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'a ⇒ ('b ⇒ 'b)" where
  "chain f [] _ = id" |
  "chain f (x#[]) y = f x y" |
  "chain f (x1#x2#xs) y = (chain f (x2#xs) y) o (f x1 x2)"
```

A function $\mathsf{chain}\, e\ xs\ y$ for a path $xs = [x_1, \ldots, x_n]$ to a designated node y represents the composition of all functions $e\ x_i\ x_{i+1}$ and $e\ x_n\ y$.

Note that the definition is more complicated than required: we could discard the second case. However, we will have to express nilpotent function chains and therefore we have to circumvent the general use of the non-nilpotent identity function in our definition. We will have to make sure in this context that no chains over empty paths occur such that the first case is not encountered.

In the above definition we use a designated destination node y to simplify splitting and concatenating path functions: the head node of the second path segment becomes the destination node for the first chain:

```
lemma chain_append:
  assumes "ys ≠ []"
  shows "chain f (xs @ ys) z = (chain f ys z) o (chain f xs (hd ys))"
```

A variant of that lemma allows us to omit the assumption that the second path segment `ys` is non-empty. However, that variant is more verbose due to the "search" for the destination node $\mathsf{hd}\,(xs \cdot [y])$ along the path:

```
lemma chain_append_nonempty:
  "chain f (xs @ ys) z a = chain f ys z (chain f xs (hd (ys @ [z])) a)"
```

This variant uses a different style by including argument `a` to the resulting function instead of expressing the resulting function using function composition. This style turned out to be superior in terms of usability in our proofs, e.g. in explicit substitutions or rule applications in extensive calculations.

Interestingly, a variant chains2 of chains also stating source nodes explicitly turned out to decrease usability. Therefore, in cases where a designated source node x, a path ys, and a destination node z are given, we usually use $\mathsf{chain}\ e\ (x \cdot ys)\ z$ instead of the deprecated function `chains2`.

To prove that chains are contained in closed sets of endomorphisms we provide several lemmas, e.g.:

```
lemma endo_chain_closed:
  assumes "∀x y. f x y ∈ E" "End_closed E" "id ∈ E"
  shows "chain f xs y ∈ E"
```

```
lemma endo_chain_closed_nonempty:
  assumes "xs ≠ []" "∀x y. f x y ∈ E" "End_closed E"
  shows "chain f xs y ∈ E"
```

The later of those two exemplary lemmas is one that is suited and designed for the nilpotent context as it avoids chain 's case for empty lists that would introduce the non-nilpotent identity function into our chain and cause the unsatisfiable assumption `nilpotent E` ∧ `id ∈ E`.

Sometimes, we will need to exchange the edge functions of chains with equivalent ones, e.g. when we have to consider a path $xs \subseteq G$ in the context of a supergraph $G' \supseteq G$ (the supergraph G' restricted to the domain of G is identical to its subgraph G, therefore we can substitute G for G'):

```
lemma chain_cong:
  assumes "f1 = f2 on set ns"
  shows "chain f1 ns n = chain f2 ns n"
```

To support chains along convoluted paths we now introduce the function

```
fun chains :: "('a ⇒ 'a ⇒ 'b ⇒ 'b) ⇒ 'a list list ⇒ 'a ⇒ ('b ⇒ 'b)" where
  "chains f [] _ = id" |
  "chains f (xs#[]) y = chain f xs y" |
  "chains f (xs1#xs2#xss) y = (chains f (xs2#xss) y) o (chain f xs1 (hd xs2))"
```

which simply concatenates a chain for each path segment of the convoluted path. Similar to simple chains there are analogous lemmas regarding chains being contained in closed sets of endomorphisms, and splitting and appending chains. Analogously to `chain` we again had to incorporate the constraint imposed by nilpotency in our definition.

The next level of abstraction that we will encounter is the capacity between two nodes of a graph:

```
fun cap'
  :: "nat ⇒ 'n set ⇒ ('n ⇒ 'n ⇒ 'm ⇒ 'm) ⇒ 'n ⇒ 'n ⇒
      ('m ⇒ 'm::cancel_comm_monoid_add)"
where
  "cap'      0  N e n n' = δ n n'" |
  "cap' (Suc i) N e n n' = δ n n' + (∑n'' ∈ N. (e n'' n') o (cap' i N e n n''))"

abbreviation "cap N ≡ cap' (card N) N"
```

where

```
definition δ :: "'n ⇒ 'n ⇒ 'm ⇒ 'm::cancel_comm_monoid_add" where
"δ ≡ λn n'. λx. if n = n' then x else 0"
```

Intuitively, capacities summarize all chains of length $<$ `i` within a given set of nodes `N` (including cycles) from some node `n` to some node `n'`. The following lemma captures exactly this intuition:

```
lemma cap'_unrolled_closed:
  assumes "∀x y. e x y ∈ E" "finite N" "n ∈ N" "End_closed E"
  shows "cap' i N e n n' = δ n n' + (∑ns ∈ l_lists' N i. chain2 e n ns n')"
```

where we use the following definition of `l_lists'`:

```
definition "l_lists' X k ≡ { xs. set xs ⊆ X ∧ length xs < k }"
```

Other kinds of lists that we will have to deal with later are:

```
definition "l_lists  X k ≡ { xs. set xs ⊆ X ∧ length xs < k ∧ length xs ≥ 1 }"

definition "k_lists  X k ≡ { xs. set xs ⊆ X ∧ length xs = k }"
```

An observation that is relevant many times in our formalization is that all of the just mentioned sets of lists are finite if their carrier sets `X` are finite.

The definition of `cap'_unrolled_closed` exhibits one of the occurences of the deprecated function `chain2`. Of course we also provide some membership lemmas for capacities regarding closed sets of endomorphisms.

The last level of abstraction that we will encounter are chains of capacities. We use a generalized variant of `chains` to express this:

```
fun chains' :: "('a list ⇒ 'a ⇒ 'b ⇒ 'b) ⇒ 'a list list ⇒ 'a ⇒ ('b ⇒ 'b)" where
  "chains' f [] _ = id" |
  "chains' f (xs#[]) y = f xs y" |
  "chains' f (xs1#xs2#xss) y = (chains' f (xs2#xss) y) o (f xs1 (hd xs2))"
```

While `chains` concatenates path functions generated by chain for each path segment, `chains'` instead allows us to specify arbitrary functions `f` to be applied on each path segment. For a given convoluted path xss we can now state the sum of all path functions that connect some sequence of common nodes[1] $[\mathsf{hd}\, xs_1, \ldots, \mathsf{hd}\, xs_n, y]$ where the path segments between common nodes are of length less than i as

$$\mathsf{chains'}\,(\lambda xs\, y.\ \mathsf{cap}'\, i\, N\, e\,(\mathsf{hd}\, xs)\, y)\, xss\, y$$

If the lengths of all path segments in xss are less than i then $\mathsf{chain}\, e\ (\mathsf{concat}\, xss)\ y$ is one of the path functions generated by the capacity chain. In particular, we then can approximate in the context of positive monoids:

$$\mathsf{chains}\, e\ xss\ y \leq \mathsf{chains'}\,(\lambda xs\, y.\ \mathsf{cap}'\, i\, N\, e\,(\mathsf{hd}\, xs)\, y)\, xss\, y$$

3.4. The Generalized Pigeonhole Principle

A concept that is quite often employed in the proofs of our formalization of the Flow Framework is the pigeonhole principle. Isabelle already provides the simple variant of the principle (for $|N| + 1$ elements drawn from a set N, there must be an element that occurs at least twice). However, Isabelle lacked[2] the generalized variant [Steger, 2007]:

```
lemma generalized_pigeonhole:
  assumes "dom f ≠ {}" "finite (dom f)" "card (dom f) ≥ card (ran f) * k + 1"
  shows "∃y ∈ ran f. card (f -` {Some y}) ≥ k + 1"
```

The reasoning that involves the simple variant of the pigeonhole principle usually follows the following pattern: we have some path whose length is larger than the number of nodes in the containing subgraph. Therefore, there must be some node in the path that occurs at least twice. Hence, the path contains a cycle. In the case of so-called effectively acyclic flow graphs we then can immediately infer that the flow along this cycle becomes zero. As we only apply endomorphisms within the context of effectively acyclic flow graphs the path's suffix following the cycle will not introduce new flow, i.e. the flow along the entire path is zero.

[1] Here, we define common nodes as nodes that occur in each path; in each path they occur in the order provided by the sequence; the first and last common node are the start and end node of each path.

[2] By now there should be a variant of the generalized pigeonhole principle that is part of Isabelle's standard library, for reasons of not wasting our work in this thesis we use our own variant.

In the case of nilpotent edge functions (with nilpotency index p) we will apply the generalized pigeon hole principle to find $p + 1$ occurrences of a node in a path and then approximate all subchains between those occurrences such that we obtain p concatenations of the same term. Nilpotency then tells us that the p identical terms equal the zero function and therefore there is no flow along the considered path.

4. Formalization of the Flow Framework

This chapter both presents and formalizes the Flow Framework from [Krishna et al., 2019, Krishna, 2019]. We start in Section 4.1 with flow graphs, flow interfaces and their relationship. Then, we will have a look at how to prove the existence and uniqueness of flows in practical scenarios in Section 4.2. Finally, we prove a theoretical result in Section 4.3.

For this chapter, we fix some type $'m$ :: `cancel_comm_monoid_add`, called *flow domain*, and a type $'n$, called *node type*. Cancellative commutative monoids `cancel_comm_monoid_add` are commutative monoids that additionally satisfy the property

$$\forall x\, y\, z.\ x + y = x + z \longrightarrow y = z$$

This property provides us some limited notion of subtraction if we know that an element is contained in a sum: $(a + b) - b = a$. From this we can infer that for an equality $c = a + b$ it follows that $c - b = (a + b) - b = a$.

4.1. Flow Graphs and Flow Interfaces

4.1.1. Flow Graphs

The basic idea behind flow graphs is to consider heaps (as in heaps from Separation Logic) as abstract partial directed graphs. For example memory locations can be considered as nodes and pointers between memory locations can be considered as edges. The finite domain of such a partial graph (N, e) will usually be denoted with N :: $'n$ set , and a function providing the *edge functions* of the graph will be denoted with e :: $'n \Rightarrow 'n \Rightarrow 'm \Rightarrow 'm$ (sometimes, we will also call e edge function). The first argument of e represents source nodes of edges, the second argument represents destination nodes of edges. An edge function $e\ x\ y$ from source node x to destination node y has type $'m \Rightarrow 'm$ and is used to propagate pieces of information of type $'m$ from x to y. Note that the first argument of e is only defined partially on domain N as there can not be edges originating from nodes that are not part of a graph (but Isabelle/HOL only provides total functions, so we have to consider e to be restricted to N by convention; maps are too restrictive regarding usability in extensive calculations). In contrast, the second argument is not restricted, i.e. we allow edges from the inside of a graph to the outside of a graph. To describe edges that do not propagate information, we simply use the edge function $\lambda\_.\ 0$.

Within graphs pieces of information are propagated along edges by edge functions. During propagation each edge function may apply modifications to the propagated piece of information. Subsequently, similar to data-flow analysis, each node aggregates all incoming pieces of information into a single piece of information. The aggregated pieces of information then are propagated to neighbor nodes along outgoing edges. Assuming a function f :: $'n \Rightarrow 'm$ that maps nodes of a graph to the corresponding aggregated pieces of information, we can formalize equilibria within this model of information propagation and aggregation as an equation system called *flow equation*:

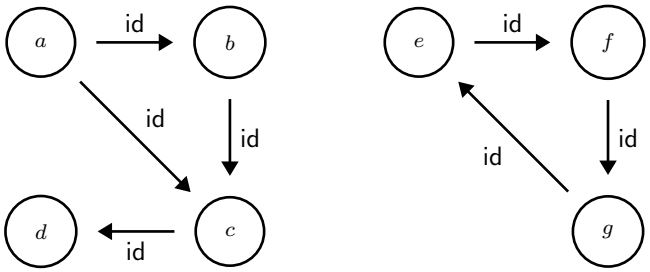
definition

```
flow_eq' :: "('n,'m) fg' ⇒ ('n ⇒ 'm::cancel_comm_monoid_add) ⇒ bool"
```

where
`"flow_eq' ≡ λ(N,e,f) i. ∀n ∈ N. f n = i n + (∑n' ∈ N. e n' n (f n'))"`

The additional function $i : {}'n \Rightarrow {}'m$ represents constant contributions of information for each node. Among other things, i will be used to model the interaction between multiple partial flow graphs (remember that we allow edges to nodes outside of a partial graph). Functions $f :: {}'n \Rightarrow {}'m$ that solve the flow equation for a given graph (N, e) and function i are called *flow* (we will use the term flow ambiguously for pieces of information of single nodes, e.g. "the flow in node n", and the propagation of pieces of information along edges and paths, e.g. "the flow along edge (x, y) or path xs"). Accordingly, we call functions i the *inflow* of a graph. Analogously to e, f is only defined on domain N of graphs (N, e). Extending the notion of graphs (N, e), we introduce the notion of *flow graphs* (N, e, f): if f is a flow for (N, e) and some inflow i then (N, e, f) is a flow graph.

Let us consider an example: given the flow domain $(\mathbb{N}, +, 0)$ and a graph $h = (N, e)$

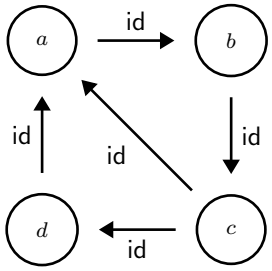


Here, we only draw edge functions that differ from $\lambda\_.\ 0$. Graph h consist of two components $\{a, b, c, d\}$ and $\{e, f, g\}$ where the second one contains a cycle. Assume we are provided two functions $i = \{a \mapsto 1, \_ \mapsto 0\}$ and $f = \{a \mapsto 1, b \mapsto 1, c \mapsto 2, d \mapsto 2, e \mapsto 3, f \mapsto 3, g \mapsto 3\}$. To show that f is a flow for graph h and inflow i we have to check for each node $n \in N \coloneqq \{a, b, c, d, e, f\}$ that

$$f\ n = i\ n = \sum_{n' \in N} f\ n \triangleright \mathsf{id} = \sum_{n' \in N} f\ n$$

In this example this obviously holds and we found a flow graph (N, e, f).

Note that our solution f is not unique: for the cycle (e, f, g) we can choose arbitrary flow values as long as the flow values are equal. If there was inflow to node e and no inflow to nodes f and g then the flow would have been unique. An example without flow is graph h



with inflow $i = \{a \mapsto 1, \_ \mapsto 0\}$. Then, for all functions f we have $f\ c = f\ b = f\ a$ and $f\ d = f\ c$ and $f\ a = 1 + (f\ c + f\ d)$. This yields the constraint $f\ a + 1 = 0$. In our flow domain there is no solution for $f\ a$.

Parts of the motivation behind the type-class constraint `cancel_comm_monoid_add` for the flow domain type ${}'m$ are induced by the flow equation: the aggregation using the sum requires

commutativity, associativity and a unit element in order to be order-independent and to cover the case that there is no edge incoming to a node. Note that the unit element was already used by the representation of edges that do not propagate information (aka. non-existing edges).

Cancellativity is relevant to be able to update flow graphs in a "local" fashion: if we do not need to propagate information from a node x (e.g. because there is no outgoing edge from x) then a change of incoming information should not imply a recomputation of the flow to and in x (potentially only bounded by the flow graph's extent) but we want to be able to handle this change locally. Cancellativity ensures that e.g. for some flow $m_1 + m_2 + m_3$ in x we can subtract contribution m_2 to x and obtain the new flow $m_1 + m_3$ in x without considering the contributors of m_1 and m_3.

4.1.2. A Type for Flow Graphs

We now start with the formalization of flow graphs. First, we introduce some preliminary type synonym $('n, 'm)\ \mathsf{fg}'$ for flow graphs that corresponds to our derivation in the previous subsection:

```
type_synonym ('n,'m) fg' = "'n set × ('n ⇒ 'n ⇒ 'm ⇒ 'm) × ('n ⇒ 'm)"
```

A value $h = (N, e, f) :: ('n, 'm)\ \mathsf{fg}'$ contains all the components that constitute a flow graph: domain N, edge functions e, flow f. However, a value h does not necessarily represent a "valid" flow graph. The above discussion yielded the following constraints for values $h = (N, e, f) :: ('n, 'm)\ \mathsf{fg}'$ to be "valid" flow graphs: first, there must be some inflow i such that the flow equation holds for h and i, and second, the domain of h is supposed to be finite. We summarize these conditions in the following predicate describing valid flow graphs within the preliminary flow graph type $('n, 'm)\ \mathsf{fg}'$:

```
definition def_fg' :: "('n,'m::cancel_comm_monoid_add) fg' ⇒ bool" where
  "def_fg' ≡ λ(N,e,f). (∃i. flow_eq' (N,e,f) i) ∧ finite N"
```

Based on the observation that e and f are only defined partially on N we additionally define an equivalence relation on flow graphs that ignores the state of e and f outside their domain:

```
definition fg'_eq :: "('n,'m) fg' ⇒ ('n,'m) fg' ⇒ bool" where
  "fg'_eq ≡ λ(N1,e1,f1) (N2,e2,f2). N1 = N2 ∧ (e1 = e2 on N1) ∧ (f1 = f2 on N1)"
```

To obtain the more pleasant notion of equality from HOL instead of having to deal with instances of `fg'_eq` we lift the preliminary flow graph type $('n, 'm)\ \mathsf{fg}'$ to a quotient type $('n, 'm)\ \mathsf{fg}$ that hides the inconvenient notion of equality of our preliminary type.

Beforehand, we have to consider an additional detail that will become relevant later when we define sums of flow graphs: sums of valid flow graphs can be invalid, i.e. addition over flow graphs is only a partial operation. This fact leaves us with two options: first, keep the sum operator partial, or second, make it total by introducing a representation for invalid flow graphs. We choose the second approach in order to obtain Isabelle's existing machinery on commutative monoids, e.g. iterated sums, for free by instantiating type-class `comm_monoid_add` for $('n, 'm)\ \mathsf{fg}$.

To obtain an explicit representation for invalid flow graphs we have to lift $('n, 'm)\ \mathsf{fg}'$ to $('n, 'm)\ \mathsf{fg}'\,\mathsf{option}$. This step is necessary as there are flow domains that do not induce invalid flow graphs (e.g. $(\{0\}, +, 0)$) that could be used to represent invalid flow graphs in the quotient type. Accordingly, our representation for invalid flow graphs within $('n, 'm)\ \mathsf{fg}'\,\mathsf{option}$ will be the value `None`. The canonical notion of equivalence for $('n, 'm)\ \mathsf{fg}'\,\mathsf{option}$ is:

```
definition fg'_option_eq
  :: "('n,'m) fg' option ⇒ ('n,'m::cancel_comm_monoid_add) fg' option ⇒ bool"
where
```

```
"fg'_option_eq ≡ λh1 h2.
  case (h1,h2) of
     (Some h1', Some h2') ⇒ if ¬def_fg' h1' ∧ ¬def_fg' h2' then True else
                            if def_fg' h1' ∧ def_fg' h2' then fg'_eq h1' h2' else
                            False
   | (None, None) ⇒ True
   | (Some h1', None) ⇒ ¬def_fg' h1'
   | (None, Some h2') ⇒ ¬def_fg' h2'"
```

In this definition, we equate `None` to `None` and all invalid preliminary flow graphs. Furthermore, all invalid preliminary flow graphs are equivalent to all other invalid preliminary flow graphs. Valid preliminary flow graphs are different from invalid ones. Finally, in the case of two defined preliminary flow graphs, we revert to the previous notion of equivalence `fg'_eq`.

Now, we obtain the actual type for flow graphs by a quotient type construction using our equivalence relation `fg'_option_eq`:

```
quotient_type (overloaded) ('n,'m) fg =
  "('n,'m::cancel_comm_monoid_add) fg' option" / fg'_option_eq
```

We denote the invalid flow graph with $\bot$:

```
lift_definition bot_fg :: "('n,'m::cancel_comm_monoid_add) fg" is None .
```

To access the components of flow graphs we introduce the following accessor functions:

```
lift_definition dom_fg :: "('n,'m::cancel_comm_monoid_add) fg ⇒ 'n set"
  is "λh. case h of Some (N,e,f) ⇒ if def_fg' (N,e,f) then N else {} | _ ⇒ {}"

lift_definition edge_fg :: "('n,'m::cancel_comm_monoid_add) fg ⇒ ('n ⇒ 'n ⇒ 'm ⇒ 'm)"
  is "λh. case h of
    Some (N,e,f) ⇒ if def_fg' (N,e,f) then restrict N (λ_ _. 0) e else (λ_ _ _. 0) |
    None ⇒ (λ_ _ _. 0)"

lift_definition flow_fg :: "('n,'m::cancel_comm_monoid_add) fg ⇒ ('n ⇒ 'm)"
  is "λh. case h of
    Some (N,e,f) ⇒ if def_fg' (N,e,f) then restrict N 0 f else (λ_. 0) |
    None ⇒ (λ_. 0)"
```

These accessor functions map the components of invalid flow graphs to default values, and pass-through the requested component for valid flow graphs. In principle, setting default values for edge functions and flow functions of valid flow graphs outside their domain is not necessary but enabled minor shortcuts in some proofs and avoids additional assumptions in some lemmas.

To specify flow graphs we introduce a constructor function:

```
lift_definition fg
  :: "'n set ⇒ ('n ⇒ 'n ⇒ 'm ⇒ 'm) ⇒ ('n ⇒ 'm) ⇒
      ('n,'m::cancel_comm_monoid_add) fg"
  is "λN e f. Some (N,e,f)" .
```

To keep the presentation simple in this thesis we will use the notation (N, e, f) synonymously to $\mathsf{fg}\, N\, e\, f$. We also will use $\mathsf{dom}\, h$ instead of `dom_fg` h, $\mathsf{edge}\, h$ instead of `edge_fg` h, and $\mathsf{flow}\, h$ instead `flow_fg` h.

To gain access to the implicitly kept flow equation of a flow graph and to prove that some constructed flow graph is a valid flow graph we provide elimination and introduction rules `fgE` and `fgI`, e.g.

```
lemma fgI:
  assumes "f = (λn. i n + (∑n' ∈ N. e n' n (f n'))) on N" "finite N"
  shows "fg N e f ≠ bot"
```

To prove equality of flow graphs there are lemmas `fg_cong`, `fg_eqI` and `fg_eqI2`, e.g.

```
lemma fg_eqI:
  assumes "h1 ≠ bot" "h2 ≠ bot" "dom_fg h1 = dom_fg h2"
    "edge_fg h1 = edge_fg h2 on dom_fg h1" "flow_fg h1 = flow_fg h2 on dom_fg h2"
  shows "h1 = h2"
```

The first lemma is for constructed flow graphs only and enables us to prove their equality on the HOL-level more conveniently as edge function and flow are only partially defined and we do not want to care about the irrelevant parts of these functions. This lemma will be used primarily in applications of the flow framework. The other two lemmas work for flow graphs in general and will be used in proofs of general properties of flow graphs, e.g. associativity.

A first result on inflows that justifies our previous statements about "the inflow" of flow graphs: the inflow of a flow graph is unique [Krishna et al., 2020, Lemma 1] (in our case restricted to the domain of the flow graph).

```
lemma flow_eq_unique:
  assumes "flow_eq h i1" "flow_eq h i2"
  shows "i1 = i2 on (dom_fg h)"
```

Another result that we will encounter multiple times is unrolling the flow equation:

```
lemma unroll_flow_eq':
  assumes "l ≥ 1" "flow_eq (fg N e f) i" "∀x y. e x y ∈ E" "End_closed E" "n ∈ N" "finite
N"
    "(λ_. 0) ∈ E"
  shows "f n = i n + (∑ns ∈ l_lists N l. (chain e ns n) (i (hd ns)))
                   + (∑ns ∈ k_lists N l. (chain e ns n) (f (hd ns)))"
```

Remember the definitions of `l_lists` and `k_lists`:

```
definition "l_lists  X k ≡ { xs. set xs ⊆ X ∧ length xs < k ∧ length xs ≥ 1 }"

definition "k_lists  X k ≡ { xs. set xs ⊆ X ∧ length xs = k }"
```

Using this lemma we can represent the flow equation as a sum over paths of up to a certain length. We obtain this representation by repeatedly unfolding the flow equation for `f` and simplifying the equation.

Later in the context of nilpotent and effectively acyclic flow graphs we will be able to eliminate the second big-sum for sufficiently far unrolled flow equations such that we obtain non-recursive representations of the flow equation.

4.1.3. The Flow Graph Algebra

As heaps are partially defined in Separation Logic, the Flow Framework's idea of representing heaps as abstract directed graphs has to reflect this property, too. This property is introduced into the Flow Framework by considering partial abstract directed graphs that can be composed and decomposed analogously to heaps in Separation Logic using the separating conjunction. In fact, the flow graph algebra we are going to construct is a separation algebra [Calcagno et al., 2007] (if restricted to valid flow graphs only). Also, the inflow will become relevant in this context to account for the interaction between separate flow graphs.

In this subsection, we show that $(`n, `m)$ `fg` in conjunction with the sum operator

```
definition
  plus_fg :: "('n,'m) fg ⇒ ('n,'m) fg ⇒ ('n,'m :: cancel_comm_monoid_add) fg"
where
  "plus_fg h1 h2 ≡
    let N = dom_fg h1 ∪ dom_fg h2;
        e = combine (dom_fg h1) (dom_fg h2) (λ_ _. 0) (edge_fg h1) (edge_fg h2);
        f = combine (dom_fg h1) (dom_fg h2)        0  (flow_fg h1) (flow_fg h2) in
    if h1 ≠ bot ∧ h2 ≠ bot ∧ dom_fg h1 ∩ dom_fg h2 = {}
      then fg N e f
      else bot"
```

and the unit element

```
definition zero_fg where
  "zero_fg ≡ fg {} (λ_ _ _. 0) (λ_. 0)"
```

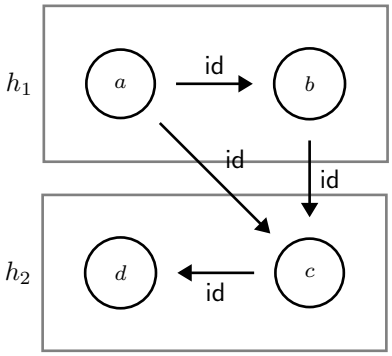
is a commutative monoid, and obtain all of Isabelle's existing results for commutative monoids. Additionally, we will show that this monoid is cancellative for valid flow graphs.

However, we will not be able to show that our monoid is a separation algebra: separation algebras are partial cancellative commutative monoids [Calcagno et al., 2007]. In principle, our algebra is even stronger than a separation algebra. Unfortunately, by including the bottom element we are unable to specify an binary operator $h_1 \# h_2$ required by the separation algebra framework that determines if $h_1 + h_2$ is defined, i.e. $h_1 \# h_2 \longleftrightarrow h_1 + h_2 \neq \bot$. In particular, one assumption that must be shown for a separation algebra is that $h \# 0$ for all $h :: (`n, `m)$ `fg`. However, for $h = \bot$ it holds that $h + 0 = \bot$ while we are required to show $h + 0 \neq \bot$. For the subtype $\{h :: (`n, `m)$ `fg` $\mid h \neq \bot\}$ of valid flow graphs it is easy to prove using our coming development that this subtype is a separation algebra (see theory `Flow_Graph_Separation_Algebra`).

Regarding this issue, it might have been advantageous to not introduce an explicit representation for the invalid flow graph and to instead rely on an unary operator, e.g. `def_fg`, that determines if a flow graph is valid. In hindsight, we do something analogous anyways using the bottom element, e.g. $h_1 + h_2 + h_3 \neq \bot$. But then we would not have been able to instantiate type class `comm_monoid_add` for $(`n, `m)$ `fg` which provides us a lot of useful infrastructure. In our case our approach is probably preferable as our choice of a Separation Logic framework for Isabelle [Lammich and Meis, 2012] does not support separation algebras. In other contexts the alternative approach might be preferable, e.g. when using Iris which supports multiple so-called resource algebras to be instantiated and used in parallel [Jung et al., 2018].

An important observation regarding `plus_fg` is that sums $h_1 + h_2$ of valid flow graphs $h_1, h_2 \neq \bot$ are not necessarily valid: if $\mathsf{dom}\, h_1 \cap \mathsf{dom}\, h_2 \neq \emptyset$ then by definition $h_1 + h_2 = \bot$. This definition and therefore the justification for this condition is founded in our close relationship with heaps and the separating conjunction from Separation Logic. The second (non-trivial) source of a sum becoming $\bot$ is hidden in the flow graph constructor `fg`: there must be an inflow such that the composite flow is a solution of the flow equation for the composite graph and the inflow.

Let us consider an example on adding flow graphs $h_1 = (N_1, e_1, f_1)$ and $h_2 = (N_2, e_2, f_2)$ for domains $N_1 = \{a, b\}$ and $N_2 = \{c, d\}$, inflows $i_1 = \{a \mapsto 1, b \mapsto 0\}$ and $i_2 = \{c \mapsto 2, d \mapsto 0\}$, and $f_1 = \{a \mapsto 1, b \mapsto 1\}$ and $f_2 = \{c \mapsto 2, d \mapsto 2\}$ (for edges see diagram below). By the definition of graph sums we have $h = h_1 + h_2 = (N, e, f)$ for $N = N_1 \cup N_2$ and $f = f_1 \cup f_2$:



We have to show that there is some inflow i such that (N, e, f) and i solve the flow equation. The solution here is $i = \{a \mapsto 1, \_ \mapsto 0\}$.

Consider the same example graph but this time with $i'_2 = \{c \mapsto 1, d \mapsto 0\}$ and $f'_2 = \{c \mapsto 1, d \mapsto 1\}$. Then we have for $h' = h_1 + h'_2 = (N', e', f')$ with $f' = f_1 \cup f'_2$ that $1 = f'_2\ c = i'\ n + (f_1\ a + f_1\ b) \geq 2$. There is no solution for this inequality in our flow domain and therefore $h_1 + h'_2 = \bot$.

As a small prospect: the reason why this example fails is that h'_2 does not expect the "amount" of inflow to node c that h_1 actually sends to c. Informally, there is some upper bound of flow that h_2 and h'_2 can accept from h_1. In the first case this upper bound is obeyed, in the second case it is violated. The reason why "less" flow than the upper bound is accepted is simply that in this case the discrepancy can be accounted as inflow of the composite graph. In contrast, "too much" flow can not be made disappear.

Another important observation that follows immediately from the definition of `plus_fg` is that for $(N, e, f) = (N_1, e_1, f_1) + (N_2, e_2, f_2) \neq \bot$ it holds that $e_k = e$ on N_k and $f_k = f$ on N_k for $k = 1, 2$ (see `edge_fg_plus_fg` and `flow_fg_plus_fg`). Furthermore, for $h = h_1 + h_2 \neq \bot$ it follows that $h_1 \neq \bot$ and $h_2 \neq \bot$ (`plus_fg_ops_exist`), $\mathsf{dom}\, h = \mathsf{dom}\, h_1 \cup \mathsf{dom}\, h_2$ (`plus_fg_dom_un`) and $\mathsf{dom}\, h_1 \cap \mathsf{dom}\, h_2 = \emptyset$ (`plus_fg_dom_disj`).

After this little detour, we return to proving the required properties for $(`n, `m)$ `fg` to be a `cancel_comm_monoid_add`. The unit element being neutral and commutativity follow immediately from the definitions of the sum operator and the unit element, as well as the fact that the unit element is trivially a valid flow graph. Proving associativity is more interesting as we have to construct an inflow for a subgraph such that the flow of the subgraph still is a solution to the flow equation.

In order to prove the first direction of associativity we will use the following split lemma that allows us to partition flow graphs into two valid flow graphs that sum up to the original one:

```
lemma split_fg:
  assumes "h ≠ bot" "dom_fg h = N1 ∪ N2" "N1 ∩ N2 = {}"
  shows "∃h1 h2. h = h1 + h2 ∧ h1 ≠ bot ∧ h2 ≠ bot ∧
    dom_fg h1 = N1 ∧ dom_fg h2 = N2 ∧
    edge_fg h = edge_fg h1 on N1 ∧ edge_fg h = edge_fg h2 on N2 ∧
    flow_fg h = flow_fg h1 on N1 ∧ flow_fg h = flow_fg h2 on N2"
```

Proof. As $h = (N, e, f) \neq \bot$ by assumption we obtain some inflow i such that the flow equation holds for h and i:

$$\forall x \in N.\ f\ x = i\ x + (\sum_{x' \in N} f\ x' \triangleright e\ x'\ x) \tag{4.1}$$

To construct the required flow graphs $h_1 = (N_1, e, f)$ and $h_2 = (N_2, e, f)$ we have to provide inflows i_1 and i_2, respectively, such that f solves the respective flow equations for h_k and i_k. For

$k = 1, 2$ we define the respective inflows as

$$\forall x \in N_k.\ i_k\ x = i\ x + (\sum_{x' \in N_{3-k}} f\ x' \triangleright e\ x'\ x) \tag{4.2}$$

and show for $x \in N_k$:

$$\begin{aligned} f\ x &= i\ x + (\sum_{x' \in N} f\ x' \triangleright e\ x'\ x) && \text{by Equation (4.1)} \\ &= i\ x + (\sum_{x' \in N_k} f\ x' \triangleright e\ x'\ x) + (\sum_{n' \in N_{3-k}} f\ x' \triangleright e\ x'\ x) && \text{by } N = N_k \dot\cup N_{3-k} \\ &= i_k\ x + (\sum_{x' \in N_k} f\ x' \triangleright e\ x'\ x) && \text{by Equation (4.2)} \end{aligned}$$

Therefore, f is a solution to both instances of the flow equation that we have to show. Furthermore, N_k is finite because $N_k \subseteq N$ and N is finite due to $h \neq \bot$. Then `fgI` provides us that $(N_k, e, f) \neq \bot$. With `plus_fg_fg'` we obtain that $(N_1, e, f) + (N_2, e, f) = (N, e, f)$. All postulated equalities of edge functions and flows of the h_k hold trivially. ■

Using `split_fg` we can provide a concise proof of the first direction of associativity that avoids unrolling the definitions of nested sums:

```
lemma plus_fg_assoc:
  fixes a b c :: "('a,'b :: cancel_comm_monoid_add) fg"
  assumes "a + b + c ≠ bot"
  shows "a + b + c = a + (b + c)"
```

Proof. We know by assumption that $(a + b) + c \neq \bot$, therefore $a + b \neq \bot$, $a \neq \bot$, $b \neq \bot$, $c \neq \bot$, and the domains of a, b, c are disjoint. We then obtain some h_1 and h_2 with $a + b + c = h_1 + h_2$ by using `split_fg` for $N_1 = \mathsf{dom}\, a$ and $N_2 = \mathsf{dom}\, b \cup \mathsf{dom}\, c$. Another application of `split_fg` for $N_1 = \mathsf{dom}\, b$ and $N_2 = \mathsf{dom}\, c$ then yields some $h_{2,1}$ and $h_{2,2}$ with $h_2 = h_{2,1} + h_{2,2}$. Using the accompanying equalities of flows and edge functions provided by `split_fg` for h_1, h_2, $h_{2,1}$ and $h_{2,2}$ we can infer that $a = h_1$, $b = h_{2,1}$ and $c = h_{2,2}$ using `fgI`. With $a + b + c = h_1 + h_2$ and $h_2 = h_{2,1} + h_{2,2}$ we obtain $a + b + c = a + (b + c)$. ■

Assuming $a + (b + c) \neq \bot$ we can show the second direction of associativity by shuffling the sum by repeated application of the first direction of associativity and commutativity:

$$a + (b + c) = (b + c) + a = b + (c + a) = (c + a) + b = c + (a + b) = (a + b) + c$$

Both directions of associativity can be combined into general associativity $(a+b)+c = a+(b+c)$ by using `fg_eqI2`, i.e. without assuming $(a + b) + c \neq \bot$ or $a + (b + c) \neq \bot$,

The first direction of cancellativity for valid flow graphs is an immediate conclusion from the equality of edge functions and flows on subdomains of sums:

```
lemma plus_fg_cancel_left:
  fixes h1 h2 h3 :: "('n,'m :: cancel_comm_monoid_add) fg"
  assumes "h1 + h2 ≠ bot"
    and "h1 + h2 = h1 + h3"
  shows "h2 = h3"
```

Proof. We apply `fg_eqI`. $h_1 + h_2 = h_1 + h_3$ and $h_1 + h_2 \neq \bot$ imply that $\mathsf{dom}\, h_2 = \mathsf{dom}\, h_3$, and that $h_2 \neq \bot$ and $h_3 \neq \bot$. The equalities $\mathsf{edge}\, h_2 = \mathsf{edge}\, h_3$ and $\mathsf{flow}\, h_2 = \mathsf{flow}\, h_3$ on $\mathsf{dom}\, h_2$ follow directly from `edge_fg_plus_fg` and `flow_fg_plus_fg` being applied to both $h_1 + h_2$ and $h_1 + h_3$. ■

The second direction (`plus_fg_cancel_right`) immediately follows from the first one together with commutativity. Note that for $(`n, `m)$ fg cancellativity only holds for valid sums: for $h_1 + h_2 = h_1 + h_3 = \bot$ it does not follow that $h_2 = h_3$, e.g. for $a \neq b$, $\mathsf{dom}\, h_1 = \{a, b\}$, $\mathsf{dom}\, h_2 = \{a\}$, $\mathsf{dom}\, h_3 = \{b\}$ it obviously holds that $h_1 + h_2 = \bot = h_1 + h_3$ due to $\mathsf{dom}\, h_1 \cap \mathsf{dom}\, h_2 \neq \emptyset \neq \mathsf{dom}\, h_1 \cap \mathsf{dom}\, h_3$, while $h_2 \neq h_3$ due to $\mathsf{dom}\, h_2 \neq \mathsf{dom}\, h_3$.

4.1.4. Flow Interfaces

In the last subsection we observed that adding two valid flow graphs might result in an invalid flow graph, e.g. due to non-disjoint domains. Aside the trivial side-conditions on disjoint and finite domains the far more critical aspect for a sum of flow graphs to be valid is that there exists an appropriate inflow for the composed graph and flow function. We now derive a characterization of valid flow graph sums.

Consider the flow equations of two flow graphs $h_1 = (N_1, e_1, f_1) \neq \bot$ and $h_2 = (N_2, f_2, e_2) \neq \bot$ with inflows i_1 and i_2, respectively, and $N_1 \cap N_2 = \emptyset$:

$$\forall n \in N_k.\ f_k\ n = i_k\ n + \sum_{n' \in N_k} f_k\ n' \triangleright e_k\ n'\ n \qquad \text{for } k = 1, 2$$

If we compare these flow equations with the flow equation of a valid sum $h_1 + h_2 = h = (N, e, f)$, where $e = \lambda x.\ \texttt{if}\ x \in N_1\ \texttt{then}\ e_1\ x\ \texttt{else}\ e_2\ x$ and $f = \lambda x.\ \texttt{if}\ x \in N_1\ \texttt{then}\ f_1\ x\ \texttt{else}\ f_2\ x$ and i is the inflow to h, then for all $k = 1, 2$ (note that in the following for a given index k we will denote the "the other" index by $3 - k$) and $n \in N_k$

$$\begin{aligned}
f_k\ n &= f\ n && \text{by def. of } f \text{ and } N_1 \cap N_2 = \emptyset \\
&= i\ n + \sum_{n' \in N} f\ n' \triangleright e\ n'\ n && \text{by flow eq.} \\
&= i\ n + (\sum_{n' \in N_k} f\ n' \triangleright e_k\ n'\ n) + (\sum_{n' \in N_{3-k}} f\ n' \triangleright e_{3-k}\ n'\ n) && \text{by } N_1 \cap N_2 = \emptyset \wedge N = N_1 \cup N_2 \\
&= i\ n + (\sum_{n' \in N_k} f_k\ n' \triangleright e_k\ n'\ n) + (\sum_{n' \in N_{3-k}} f_{3-k}\ n' \triangleright e_{3-k}\ n'\ n) && \text{by def. of } f
\end{aligned}$$

it follows by cancellativity that for $n \in N_k$ it must hold that

$$i_k\ n = i\ n + \sum_{n' \in N_{3-k}} f_{3-k}\ n' \triangleright e_{3-k}\ n'\ n$$

If $h_1 + h_2$ is not a valid sum then we obtain analogously to the above derivation that for some $k \in \{1, 2\}$ and $n \in N_k$ it must hold that

$$f_k\ n \neq i\ n + (\sum_{n' \in N_k} f_k\ n' \triangleright e_k\ n'\ n) + (\sum_{n' \in N_{3-k}} f_{3-k}\ n' \triangleright e_{3-k}\ n'\ n)$$

and therefore

$$i_k\ n \neq i\ n + \sum_{n' \in N_{3-k}} f_{3-k}\ n' \triangleright e_{3-k}\ n'\ n$$

Hence, combining both cases we obtain a characterization of valid sums:

$$h_1 + h_2 \neq \bot \longleftrightarrow \left(\forall k \in \{1, 2\}, n \in N_k.\ i_k\ n = i\ n + \sum_{n' \in N_{3-k}} f_{3-k}\ n' \triangleright e_{3-k}\ n'\ n \right) \tag{4.3}$$

The sum in Equation (4.3) exactly corresponds to the inflow to h_k: the first summand is the inflow i to h and the second summand is the contribution from h_{3-k} to h_k. As h_k is now considered on its own separated from h_{3-k} the contribution from h_{3-k} to h_k becomes an inflow to h_k instead of being propagated internally within h as flow, i.e. the original contribution from h_{3-k} to h_k via flow is simply relabeled as inflow. For this reason we introduce the notion of *outflow* of a flow graph (N, e, f):

$$\forall n \in -N.\ \mathsf{outf}\,(N,e,f)\ n = \sum_{n' \in N} f\ n' \triangleright e\ n'\ n$$

Substituting this new notation into Equation (4.3) we obtain the following characterization of valid sums:

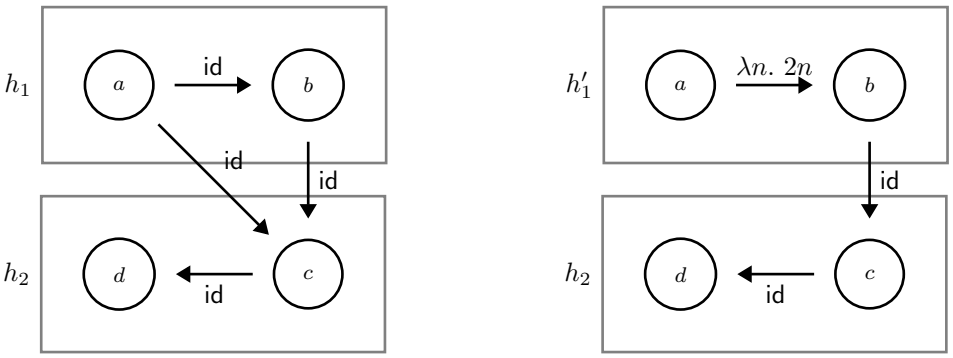
$$h_1 + h_2 \neq \bot \longleftrightarrow (\forall k \in \{1,2\}, n \in N_k.\ i_k\ n = i\ n + \mathsf{outf}\ h_{3-k}\ n) \tag{4.4}$$

We note that this characterization provides us some degree of freedom: it only depends on the *interface*, i.e. inflow and outflow, of a flow graph. Therefore, composability of flow graphs does not immediately depend on the internal structure of flow graphs but only on the implied interface. This is a weaker notion than equality of flow graphs: for example the outflow sum does not distinguish between different paths the outflow might take as long as the total outflow is the same.

Let us consider an opportunity to exploit this observation: if there is a flow graph $h_1' = (N_1', e_1', f_1')$ with inflow i_1' and both $i_1' = i_1$ and $\mathsf{outf}\ h_1' = \mathsf{outf}\ h_1$ hold, then with $h_2' \coloneqq h_2$

$$\begin{aligned} h_1 + h_2 \neq \bot &\longleftrightarrow (\forall k \in \{1,2\}, n \in N_k.\ i_k\ n = i\ n + \mathsf{outf}\ h_{3-k}\ n) \\ &\longleftrightarrow \left(\forall k \in \{1,2\}, n \in N_k.\ i_k'\ n = i\ n + \mathsf{outf}\ h_{3-k}'\ n\right) \\ &\longleftrightarrow h_1' + h_2 \neq \bot \end{aligned}$$

Reconsider our previous example on the left hand side with $i_1 = \{a \mapsto 1, b \mapsto 0\}$, $i_2 = \{c \mapsto 2, d \mapsto 0\}$, $f_1 = \{a \mapsto 1, b \mapsto 1\}$ and $f_2 = \{c \mapsto 2, d \mapsto 0\}$:

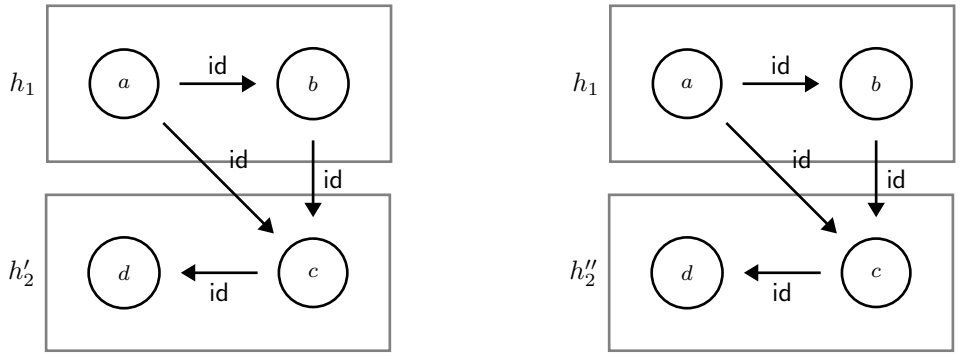


i_1 is the inflow of h_1 and $o_1\ n = \sum_{n' \in \{a,b\}} f_1\ n' \triangleright e_1\ n'\ n$ is the outflow of h_2. We can compute that $o_1 = \{c \mapsto 2, d \mapsto 0\}$. On the right hand side we have an obviously different second flow graph $h_1' + h_2$. For inflow $i_1' \coloneqq i_1$ and $f_1' = \{a \mapsto 1, b \mapsto 2\}$ we have outflow $o_1' = o_1$. According to our characterization Equation (4.4) we obtain $h_1' + h_2 \neq \bot$ by only checking the equalities $i_1' = i_1$ and $o_1' = o_1$.

The above observations induce a congruence relation on flow graphs with identical interfaces: in sums each flow graph can be replaced by an equivalent one without running danger that a

previously valid sum becomes invalid. Therefore, if we modify a sub-flow graph and preserve the interface of the sub-flow graph then we can directly recompose the modified sub-flow graph with its sibling sub-flow graph without explicitly proving the existence of a flow for the resulting graph.

The characterization also explains our informal explanation why one of our examples on the addition of flow graphs did not work:



For $i_1 = \{a \mapsto 1, b \mapsto 0\}$, $f_1 = \{a \mapsto 1, b \mapsto 1\}$, $i_2' = \{c \mapsto 1, d \mapsto 0\}$, $f_2' = \{c \mapsto 2, d \mapsto 2\}$ we have outflows $o_1 = \{c \mapsto 2, d \mapsto 0\}$ and $o_2 = \{\_ \mapsto 0\}$. When we try to calculate the inflow of $h_1 + h_2'$ to c then we obtain $1 = i_2\ c = i\ c + o_1\ c = i\ c + 2$ which is not solvable as $i\ c \geq 0$. In contrast, for $i_2'' = \{c \mapsto 3, d \mapsto 0\}$ and $f_2'' = \{c \mapsto 3, d \mapsto 3\}$ we obtain solution $i\ c = 1$ for $3 = i_2''\ c = i\ c + o_1\ c = o_1\ c + 2$.

4.1.5. A Type for Flow Interfaces

We now develop an explicit representation for flow interfaces, i.e. an explicit representation of the just described congruence classes of flow graphs. This development proceeds analogously to the one of flow graphs: we create a preliminary type for flow interfaces, introduce some quotient type for actual flow interfaces based on some notion of equality and validity.

The preliminary type for flow interfaces is

```
type_synonym ('n,'m) fi' = "'n set × ('n ⇒ 'm) × ('n ⇒ 'm)"
```

A flow interface value $(N, i, o) :: (`n, `m)\ \mathsf{fi}'$ is composed from three components: domain N, inflow i, and outflow o.

Two flow interfaces (N_1, i_1, o_1) and (N_2, i_2, o_2) are equal iff. $N_1 = N_2$, i_1 and i_2 agree on N, and o_1 and o_2 agree on $-N$ (remember that inflow and outflow are defined on disjoint domains). The lifted equality for $(`n, `m)\ \mathsf{fi}'\ \mathsf{option}$ is defined canonically analogously to $(`n, `m)\ \mathsf{fg}'\ \mathsf{option}$. The only constraint imposed on flow interfaces by our flow interface validity predicate `def_fi'` is that N has to be finite.

Using these preliminary definitions we already can define the final type for flow interfaces:

```
quotient_type (overloaded) ('n,'m) fi =
  "('n,'m :: cancel_comm_monoid_add) fi' option" / fi'_option_eq
```

A constructor function `fi`, the bottom element $\bot$, accessor functions `dom_fi`, `inf_fi` and `outf_fi`, and equality lemmas are defined analogously to their flow graph counterparts. The only pitfall that needs some attention is that the outflow is defined on $-N$ and not on N. The lemma proving validity for some flow interface only has to check that the domain of the flow interface is

finite. In this thesis we use inf [1] synonymously for `inf_fi`, outf for `outf_fi` and dom for `dom_fg`. Note that dom is used ambiguously, however, it should be always obvious which one is meant.

Note that for flow interfaces we could have avoided the quotient type construction by representing flow interfaces as tuples (N, f) and defining inflow $i \coloneqq f|_N$ and outflow $o \coloneqq f|_{-N}$ due to the complementary domains of inflows and outflows. As neither variant provides significant advantage in terms of usability (e.g. both variants require accessor and constructor to encode or decode flow interface values) we chose the analogous approach to flow graphs.

4.1.6. Flow Interface Algebra

In order to define the flow interface algebra we have to formalize the intuition developed two subsections above. The implicit nature of our characterization Equation (4.4) of the validity of flow graph sums sums is captured by the following predicate that determines if two flow graphs h_1 and h_2 with interfaces i_1 and i_2, respectively, can be composed to a flow graph $h_1 + h_2$ with interface i_{12}. The constraints on the domains of the interfaces immediately follow from the corresponding constraints imposed by the flow graph sum operator, and the outflow of the flow graphs simply simply adds up [Krishna et al., 2020, def. 6]:

```
definition is_sum_fi
  :: "('n,'m) fi ⇒ ('n,'m) fi ⇒ ('n,'m::cancel_comm_monoid_add) fi ⇒ bool" where
  "is_sum_fi i1 i2 i12 ≡
    i1 ≠ bot ∧ i2 ≠ bot ∧ i12 ≠ bot ∧
    dom_fi i1 ∩ dom_fi i2 = {} ∧
    dom_fi i1 ∪ dom_fi i2 = dom_fi i12 ∧
    (∀n ∈ dom_fi i1. inf_fi i1 n = inf_fi i12 n + outf_fi i2 n) ∧
    (∀n ∈ dom_fi i2. inf_fi i2 n = inf_fi i12 n + outf_fi i1 n) ∧
    (∀n ∈ -dom_fi i12. outf_fi i12 n = outf_fi i1 n + outf_fi i2 n)"
```

We now define the plus operator for flow interfaces using the The-operator to obtain a solution to this predicate. If there is no solution then the result is $\bot$:

```
definition plus_fi
  :: "('n,'m) fi ⇒ ('n,'m) fi ⇒ ('n,'m::cancel_comm_monoid_add) fi" where
  "plus_fi i1 i2 ≡
    if (∃i12. is_sum_fi i1 i2 i12)
      then (THE i12. is_sum_fi i1 i2 i12)
      else bot"
```

To obtain access to this implicit representation we introduce two lemmas:

```
lemma plus_fi_to_is_sum_fi:
  assumes "i1 + i2 = i12" "i12 ≠ bot"
  shows "is_sum_fi i1 i2 i12"

lemma is_sum_fi_to_plus_fi:
  assumes "is_sum_fi i1 i2 i12"
  shows "i1 + i2 = i12"
```

Using these two lemmas we can directly reduce all proof obligations regarding the instantiation of `cancel_comm_monoid_add` for $('n, 'm)$ fi to `is_sum_fi`.

The unit element of flow interfacese is defined as

[1] Some proof readers complained that our choice of this symbol is too confusing regarding the infimum. Unfortunately, the consequences of changing it at this point in time are too severe due to consistency with our formalization.

```
definition zero_fi where
  "zero_fi ≡ fi {} (λ_.0) (λ_.0)"
```

Commutativity and neutrality of the unit element are trivial again. The first direction of associativity again requires some work [Krishna et al., 2020, th. 1]:

```
lemma is_sum_assoc_rl:
  assumes "is_sum_fi i2 i3 i23" "is_sum_fi i1 i23 i123"
  shows "∃i12. is_sum_fi i1 i2 i12 ∧ is_sum_fi i12 i3 i123"
```

Proof. This proof is an immediate formalization of the proof in [Krishna, 2019, th. 1]. ■

The second direction is proven using the same approach already applied for flow graphs: shuffle the sums using the first direction of associativity and commutativity until the desired form is obtained. Together, both directions can be combined to general associativity using `fi_eqE2`.

In order to prove cancellativity we show the following basically automatically proven lemma using `fi_eqI` and the definition of `is_sum_fi`:

```
lemma is_sum_fi_unique2:
  assumes "is_sum_fi i1 i2 i12" "is_sum_fi i1 i2' i12"
  shows "i2 = i2'"
```

Using this, cancellativity follows directly after converting the flow interfaces of the involved sums to `is_sum_fi` (as with flow graphs: for valid sums only):

```
lemma plus_fi_cancel_right:
  fixes a b c :: "('a,'b::cancel_comm_monoid_add) fi"
  assumes "a + b ≠ bot" "a + b = a + c"
  shows "b = c"
```

The other direction of cancellativity again follows using commutativity.

Note that in principle flow interfaces form a separation algebra (see theory `Flow_Interface-Separation_Algebra`). However, the same discussion applies as already for flow graphs.

4.1.7. Using Flow Interfaces

The previous subsection introduced flow interfaces independently from flow graphs. In this subsection we relate flow graphs and flow interfaces.

The inflow of flow graphs is only available implicitly as the function i that makes the flow of a flow graph satisfy the flow equation [Krishna et al., 2020, informally, p. 316]. In order to obtain this implicit information we apply Hilbert's Epsilon:

```
definition inf_fg :: "('n,'m) fg ⇒ ('n ⇒ 'm::cancel_comm_monoid_add)" where
  "inf_fg h ≡ let i = (SOME i. flow_eq h i) in restrict (dom_fg h) 0 i"
```

The outflow of flow graphs corresponds to our finding in Section 4.1.4 and can be stated explicitly [Krishna et al., 2020, p. 316]:

```
definition outf_fg :: "('n,'m) fg ⇒ ('n ⇒ 'm::cancel_comm_monoid_add)" where
  "outf_fg h ≡
    λn. if n ∈ dom_fg h then 0 else (∑n' ∈ dom_fg h. edge_fg h n' n (flow_fg h n'))"
```

Together, inflow and outflow form the interface of a flow graph [Krishna et al., 2020, p. 316]:

```
definition int_fg :: "('n,'m) fg ⇒ ('n,'m::cancel_comm_monoid_add) fi" where
  "int_fg h ≡ if h = bot then bot else fi (dom_fg h) (inf_fg h) (outf_fg h)"
```

In the following we will use the notation $\mathsf{inf}\, h$ synonymously to `inf_fg h`, $\mathsf{outf}\, h$ synonymously to `outf_fg` h and $\mathsf{int}\, h$ synonymously to `int_fg` h. Note that inf and outf are used ambiguously for both flow graphs and flow interfaces.

A first observation is that $\mathsf{int}\, h \neq \bot$ iff. $h \neq \bot$. A first result on the relationship between singleton flow graphs and flow interfaces is [Krishna et al., 2019, Lemma 2]:

```
lemma int_fg_singleton_fi:
  assumes "h ≠ bot" "n ∈ dom_fg h" "edge_fg h n n = (λ_. 0)"
  shows "int_fg (fg {n} (edge_fg h) (flow_fg h)) =
    fi {n} (λ_. flow_fg h n) (λn'. edge_fg h n n' (flow_fg h n))"
```

The central lemma of this subsection relates sums of flow graphs and sums of flow interfaces [Krishna et al., 2020, Lemma 2]:

```
lemma int_fg_fi_hom: "int_fg (h1 + h2) = int_fg h1 + int_fg h2"
```

Proof. This proof is an immediate formalization of the proof in [Krishna et al., 2020]. ■

This lemma exactly captures the intuition of being able to substitute flow graphs in sums if their interfaces are equal. Given some flow graph $(h_1 + h_2) + h_3 \neq \bot$. This implies that $\mathsf{int}\,((h_1 + h_2) + h_3) \neq \bot$. Applying lemma `int_fg_fi_hom` repeatedly we obtain for h_1' with $\mathsf{int}\, h_1' = \mathsf{int}\, h_1$ that

$$\begin{aligned}\mathsf{int}\,((h_1 + h_2) + h_3) &= (\mathsf{int}\, h_1 + \mathsf{int}\, h_2) + \mathsf{int}\, h_3 \\ &= (\mathsf{int}\, h_1' + \mathsf{int}\, h_2) + \mathsf{int}\, h_3 \\ &= \mathsf{int}\,((h_1' + h_2) + h_3)\end{aligned}$$

This in turn implies that $(h_1' + h_2) + h_3 \neq \bot$.

Note that equivalent flow graphs always have the same domain. Therefore, it is only possible to infer validity of flow graphs after changes to edges or flow using the above lemma. However, we are unable to extend flow graphs using the lemma. To extend flow graphs with additional nodes we introduce the additional notion of *contextual extension* [Krishna et al., 2020, definition 7]:

```
definition contextual_extension
  :: "('n,'m) fi ⇒ ('n,'m::cancel_comm_monoid_add) fi ⇒ bool" ("_ ≲ _")
where
  "contextual_extension ≡ λi1 i2. dom_fi i1 ⊆ dom_fi i2 ∧
    (inf_fi i1 = inf_fi i2 on dom_fi i1) ∧
    (outf_fi i1 = outf_fi i2 on -dom_fi i2)"
```

Contextual extension captures the fact that some interface i_2 extends some "smaller" interface i_1 in a way that allows us to substitute the "larger" interface i_2 for i_1 using the replacement theorem [Krishna et al., 2020, theorem 2]:

```
theorem replacement:
  assumes "i = i1 + i2" "i1 ≲ i1'" "i ≠ bot" "i1' ≠ bot"
    "dom_fi i1' ∩ dom_fi i2 = {}" "outf_fi i2 = (λ_. 0) on (dom_fi i1' - dom_fi i1)"
  shows "∃i'. i' = i1' + i2 ∧ i ≲ i' ∧ i' ≠ bot"
```

We provide a simpler proof than [Krishna, 2019] in our formalization. Instead of reverting to the flow graphs underlying the flow interfaces we simply define $i' = (N', in', out')$ where $N' = N_1' \cup N_2$, $in'\ n = \texttt{if}\ n \in N_1' \setminus N_1\ \texttt{then}\ in_1'\ n\ \texttt{else}\ in\ n$ and $out' = out$, and show that $i' = i_1' + i_2 \neq \bot$ and $i \preccurlyeq i'$. In Isabelle each of these goals is basically proven automatically.

A use case of this theorem might look as follows: consider a sum $(h_1 + h_2) + h_3 \neq \bot$ and suppose we want to extend h_1 with a fresh node x that does not receive inflow from any of the h_k. We first encapsulate x in a new singleton flow graph h_x and prove that $h_x \neq \bot$ (this is usually a simple task for singleton flow graphs if possible). Then we prove the statement $0 \preccurlyeq \mathsf{int}\, h_x$. Then $(h_1 + h_2) + h_3 \neq \bot$ implies $h_1 + h_2 \neq \bot$, $h_3 \neq \bot$ and $h_1 \neq \bot$. Using `replacement` we can show together with $0 + \mathsf{int}\, h_1 = \mathsf{int}\, h_1 \neq \bot$ (follows from $h_1 \neq \bot$) and $0 \preccurlyeq \mathsf{int}\, h_x$ that $\mathsf{int}\, h_x + \mathsf{int}\, h_1 \neq \bot$ and $0 + \mathsf{int}\, h_1 \preccurlyeq \mathsf{int}\, h_x + \mathsf{int}\, h_1$. Another application of `replacement` then yields together with $\mathsf{int}\, h_1 + \mathsf{int}\, h_2 \neq \bot$ (follows from $h_1 + h_2 \neq \bot$ and `int_fg_fi_hom`) and $0 + \mathsf{int}\, h_1 \preccurlyeq \mathsf{int}\, h_x + \mathsf{int}\, h_1$ that $(\mathsf{int}\, h_x + \mathsf{int}\, h_1) + \mathsf{int}\, h_2 \neq \bot$ and $(0 + \mathsf{int}\, h_1) + \mathsf{int}\, h_2 \preccurlyeq (\mathsf{int}\, h_x + \mathsf{int}\, h_1) + \mathsf{int}\, h_2$. A third iteration provides us that $((\mathsf{int}\, h_x + \mathsf{int}\, h_1) + \mathsf{int}\, h_2) + \mathsf{int}\, h_3 \neq \bot$ and $((0 + \mathsf{int}\, h_1) + \mathsf{int}\, h_2) + \mathsf{int}\, h_3 \preccurlyeq ((\mathsf{int}\, h_x + \mathsf{int}\, h_1) + \mathsf{int}\, h_2) + \mathsf{int}\, h_3$. We can conclude that $((h_x + h_1) + h_2) + h_3 \neq \bot$ using `int_fg_fi_hom` (which we already applied multiply times in this examples). In total we were able to extend h_1 to $h_1' = h_x + h_1$ and to substitute h_1' for h_1 in existing relationships while maintaining validity. Therefore, `replacement` enables us to allocate and include new nodes in our flow graphs. In practical use cases the restriction that there must not be inflow to the newly allocated nodes should not impose problems as there should not be any flow to a node that did not exist before.

4.2. Existence and Uniqueness of Flows

The issue with flow graphs is to prove that flows exist. Due to the recursive nature of solutions to the flow equation, this issue is non-trivial. [Krishna et al., 2020] identified three special cases of flow domains that enable finding solutions efficiently: edge-local flows, nilpotent cycles, effectively acyclic flow graphs. Edge-local flows break the recursive structure of the flow equation because constant edge functions omit their argument, i.e. in $f\ n = i\ n + \sum_{n' \in N} f\ n' \triangleright e\ n'\ n$ the values $f\ n' \triangleright e\ n'\ n$ are actually independent from $f\ n'$ and it is trivial to compute f. For nilpotent and effectively acyclic flow graphs we can show that all paths of a certain length do not propagate any flow. E.g. if this certain length is l then we can simplify the unrolled flow equation

$$\begin{aligned} f\ n &= i\ n + \Big(\sum_{\substack{xs \subseteq N \\ |xs| < l}} i\ n' \triangleright \mathsf{chain}\, e\ xs\ n\Big) + \Big(\sum_{\substack{xs \subseteq N \\ |xs| = l}} f\ n' \triangleright \mathsf{chain}\, e\ xs\ n\Big) \\ &= i\ n + \Big(\sum_{\substack{xs \subseteq N \\ |xs| < l}} i\ n' \triangleright \mathsf{chain}\, e\ xs\ n\Big) + \Big(\sum_{\substack{xs \subseteq N \\ |xs| = l}} 0\Big) \\ &= i\ n + \Big(\sum_{\substack{xs \subseteq N \\ |xs| < l}} i\ n' \triangleright \mathsf{chain}\, e\ xs\ n\Big) \end{aligned}$$

The simplified unrolled flow equation does not depend anymore on f but only on the fixed inflow i. Therefore, we can simply compute f. We prove all three approaches in this section.

The general approach of conducting proofs using the Flow Framework is to abstract single entities x of instances of data structures by flow interfaces i_x. The flow interfaces corresponding the entire instances of data structures then have the form $i = \sum_x i_x$. We introduce global constraints $\varphi(i)$ to the flow interfaces of instances. These constraints are immutable and must be

satisfied for the entire live-time of each instance. Global constraints φ can be chosen arbitrarily for each data structure, the only requirement is that they enable the desired properties to be proven. Additionally, we introduce entity-local invariants $\gamma(x, i_x)$ that constrain the flow interfaces i_x of single entities and also connect entities x with their interface i_x. In conjunction, all this information has to imply the property that is supposed to be proven.

This view explains the role of flow and inflow in the definition of flow graphs. In principle, there are two variable components in a flow graph that are relevant to solve the flow equation: flow and inflow. In our opinion the existential quantification of inflow in the definition of flow graphs even hints that the inflow is the parameter used to make given flows solve the flow equation. However, in contrast to this intuition our wording always suggested that the flow has to solve the flow equation and that the inflow has a kind of fixed role. This reasoning stems from the fact that usually the global inflow is fixed by a global invariant φ and this implies that according to the subdivison of the entirety of the flow graph the inflows of sub-flow graphs are already implicitly determined. Therefore, all lemmas following in this section are of the following form: given a graph (N, e) and an inflow i there exists a unique flow f for (N, e) and i.

4.2.1. Edge-Local Flows

A simple instance of proving the existence of flow for a given graph and inflow is when all edge functions are equal to a constant function. We describe sets of constant edge functions using a predicate

```
abbreviation "const E ≡ ∀e ∈ E. ∃a. e = (λ_. a)"
```

To constrain the edge functions within a flow graph to those contained in a set of functions E, like `const` E, `nilpotent` E or `End_closed` E, we apply assumptions of the following pattern:

$$\forall x\, y.\ \mathsf{edge}\, h\, x\, y \in E$$

In contrast to [Krishna et al., 2020] we use explicit representations for facts like nilpotency or constraining edge functions instead of tying the set of allowed edge functions to the flow domain. Our motivation behind the explicit representation of these facts was to avoid an explicit type for flow domains that includes the set of allowed edge functions. Instead, we directly used the existing type-class `cancel_comm_monoid_add` for our flow domain and handle sets of allowed edge functions separately from our flow domain. This approach also avoids a lot of notational overhead in the previous developments as the set of allowed edge functions is irrelevant there.

Using this predicate we can already prove the lemma about the existence and uniqueness of edge-local flows [Krishna, 2019, lem. 3.26]:

```
lemma edge_local:
  assumes "const E" "∀x y. e x y ∈ E" "N ≠ {}" "finite N"
  shows "∃f. flow_eq (fg N e f) i ∧ (∀f'. flow_eq (fg N e f') i ⟶ f = f' on N)"
```

Proof. By assumption we know that for each x and y there is some $a_{x,y}$ such that $e\ x\ y = \lambda\_.\ a_{x,y}$. We now show that $f \coloneqq \lambda n.\ i\ n + \sum_{n' \in N} a_{x,y}$ satisfies the flow equation for graph (N, e) and inflow i. For $n \in N$ we have

$$f\ n = i\ n + \sum_{n' \in N} a_{n',n} = i\ n + \sum_{n' \in N} f\ n' \triangleright e\ n'\ n$$

Therefore, the flow equation holds.

To show uniqueness we assume the flow equation holds also for flow graph (N, e, f') and i and we have to show that $f = f'$. Assume $n \in N$. Then

$$f\ n = i\ n + \sum_{n' \in N} a_{n',n} = i\ n + \sum_{n' \in N} f'\ n' \triangleright e\ n'\ n = f'\ n$$

■

4.2.2. Nilpotent Flow Graphs

The second approach assumes that all edge functions within a flow graph are nilpotent endomorphisms. We remember the definition of closed sets of endomorphisms and nilpotent sets of functions from Section 3.2:

```
definition End_closed where
  "End_closed E ≡ E ⊆ End ∧
    (∀f1 f2. f1 ∈ E ⟶ f2 ∈ E ⟶ f1 + f2 ∈ E ∧ f2 o f1 ∈ E)"

definition nilpotent :: "('a ⇒ 'a :: zero) set ⇒ bool" where
  "nilpotent E ≡ ∃p > 1. ∀f ∈ E. f^^p = (λ_. 0)"
```

Nilpotent endomorphisms then are represented by `End_closed` $E \wedge$ `nilpotent` E. Using the definition of nilpotency we are now able to formalize that there is a unique flow for any given inflow i and graph (N, e) whose edge functions are constrained to a set of nilpotent endomorphisms E [Krishna et al., 2020, lemma 4]:

```
lemma nilpotent_flow_exists:
  fixes E :: "('m ⇒ 'm :: pos_cancel_comm_monoid_add) set"
  assumes "∀x y. e x y ∈ E" "End_closed E" "nilpotent E" "finite N"
    "(λ_. 0) ∈ E" "finite N"
  shows "∃f. flow_eq (fg N e f) i ∧ (∀f'. flow_eq (fg N e f') i ⟶ f = f' on N)"
```

Lacking an explicit representation for graphs we directly state some domain N and edge functions e. The only constraints that we have to obey for these two components is that N must be finite and that all edge functions must be from E.

Proof. If $N = \emptyset$ then the statement holds trivially. Therefore, we are given a graph (N, e) with $N \neq \emptyset$ and an inflow i and have to construct a flow f such that (N, e, f) and i satisfy the flow equation. All edge functions are drawn from a set E of nilpotent endomorphisms. Let $p > 1$ denote the nilpotency index of E. We define $l \coloneqq |N| \cdot p + 1$. Due to $N \neq \emptyset$ and $p \geq 2$ it follows $l \geq 2$. Furthermore, we define a function f: for each $n \in N$

$$f\ n \coloneqq i\ n + \left(\sum_{\substack{ns \subseteq N \\ 1 \leq |ns| < l}} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n \right)$$

We prove that f solves the flow equation for (N, e, f) and i.

But first we show for all ns with $|ns| = l$ that $\mathsf{chain}\ \ e\ ns\ n = 0$. Due to $l = p \cdot |N| + 1$ and $ns \subseteq N$ the generalized pigeonhole principle (`generalized_pigeonhole`) provides us with some $m \in N$ such that m occurs $p + 1$ times in ns. Let ms and mss be the decomposition of ns such that $ns = ms \cdot \mathsf{concat}\, mss$ with $mss \neq []$ and $|mss| = p + 1$ and $\forall ms \in mss.\ \mathsf{hd}\, ms = m \wedge ms \neq [] \wedge |ms| \leq |ns|$.

We now define a function

$$g \coloneqq \sum_{\substack{xs \subseteq N \\ |xs|<|ns|}} \mathsf{chain}\, e\ (m \cdot xs)\ m$$

Then, for each $ms \in mss$ due to $|ms| \leq |ns|$, $\mathsf{hd}\, ms = m$ and positivity:

$$\mathsf{chain}\, e\ (m \cdot ms)\ m \leq g$$

From $|mss| = p{+}1$ it follows that $|\mathsf{butlast}\, mss| = p$ and from $p > 1$ it follows that $\mathsf{butlast}\, mss \neq []$. We approximate

$$\begin{aligned}
\mathsf{chain}\, e\ ns\ n &= \mathsf{chains}\, e\ (ms \cdot mss)\ n \\
&= \mathsf{chain}\, e\ ms\ m \triangleright \mathsf{chains}\, e\ (\mathsf{butlast}\, mss)\ m \triangleright \mathsf{chain}\, e\ (\mathsf{last}\, ms)\ n \\
&\leq \mathsf{chain}\, e\ ms\ m \triangleright g^p \triangleright \mathsf{chain}\, e\ (\mathsf{last}\, ms)\ n \\
&= 0
\end{aligned}$$

The first equality uses our decomposition of ns; the second one applies the definition of `chains` and lemma `chains_append1`; the inequality iteratively approximates each chain in chains with g; the last equality follows by nilpotency and $p > 1$. In the second equality it is essential that $\mathsf{butlast}\, mss \neq []$ and $\forall ms \in mss.\ ms \neq []$: otherwise $\mathsf{chains}\, e\ (\mathsf{butlast}\, mss)\ m$ would include the non-nilpotent identity function and we can not apply nilpotency in the last step.

By positivity, we obtain $\mathsf{chain}\, e\ ns\ n = 0$ for all ns with $|ns| = l$.

We now prove that f solves the flow equation for (N, e, f) and i. For notational reasons we assume in the following derivation that all sums quantifying over a variable ns also impose the implicit constraint $ns \subseteq N$.

$$\begin{aligned}
f\ n &= i\ n + (\sum_{1 \leq |ns| < l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) + (\sum_{|ns|=l} f\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) \\
&= i\ n + (\sum_{1 \leq |ns| < l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) \\
&= i\ n + (\sum_{1 \leq |ns| \leq l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) \\
&= i\ n + (\sum_{|ns|=1} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) + (\sum_{2 \leq |ns| \leq l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) \\
&= i\ n + (\sum_{|ns|=1} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) + (\sum_{n' \in N} \sum_{1 \leq |ns| < l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ (ns \cdot n')\ n) \\
&= i\ n + (\sum_{n' \in N} i\ n' \triangleright \mathsf{chain}\ \ e\ [n']\ n) + (\sum_{n' \in N} \sum_{1 \leq |ns| < l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n' \triangleright e\ n'\ n) \\
&= i\ n + (\sum_{n' \in N} i\ n' \triangleright e\ n'\ n) + (\sum_{n' \in N} \sum_{1 \leq |ns| < l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n' \triangleright e\ n'\ n) \\
&= i\ n + (\sum_{n' \in N} (i\ n' + \sum_{1 \leq |ns| < l} i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\ \ e\ ns\ n') \triangleright e\ n'\ n) \\
&= i\ n + (\sum_{n' \in N} f\ n' \triangleright e\ n'\ n)
\end{aligned}$$

The first equality unrolls the flow equation; the second one applies $\mathsf{chain}\, e\ ns\ n = 0$ to each summand of the second big sum; the third one adds all $i\ (\mathsf{hd}\, ns) \triangleright \mathsf{chain}\, e\ ns\ n = 0$ for ns with

$ns = l$ to the sum; the fourth one splits the sum according to the length of the ns; the fifth one introduces a name for $\mathsf{last}\, ns$ in the second sum and decomposes the sum; the sixth one replaces the singleton lists ns with a name for the only element in that list; the seventh one unfolds function chain applied to a singleton list; the eighth one applies distributivity; the last one substitutes the definition of f.

It remains to show that f is unique. Therefore, let f' be a second solution to the flow equation. After unrolling the flow equation for f' to paths of length of up to l we obtain using our observation that $\mathsf{chain}\, e\ ns\ n = 0$ for all ns with $|ns| = l$ and the definition of f:

$$\begin{aligned} f'\ n &= i\ n + (\sum_{1 \le |ns| < l} i\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) + (\sum_{|ns| = l} f'\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) \\ &= i\ n + (\sum_{1 \le |ns| < l} i\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\ \ e\ ns\ n) \\ &= f\ x \end{aligned}$$

■

Note that the existing proof in [Krishna et al., 2020] contains two shortcomings: first, the proof already assumes the existence of a function f such that the flow equation holds for (N, e, f) and i instead of stating a flow f and proving that it solves the flow equation. Second, the proof approximates paths from and to m using the capacity instead of our upper bound g. However, their upper bound renders the proof invalid as $\mathsf{cap}_i\ G\ m\ m = \delta_{m=m} + \ldots = \mathrm{id} + \ldots$, i.e. the upper bound contains the identity function which obviously is not nilpotent but has to be contained in the nilpotent set of endomorphisms E in order to obtain $(\mathsf{cap}_i\ G\ m\ m)^p = 0$ by nilpotency.

4.2.3. Effectively Acyclic Flow Graphs

Another approach to proving the existence and uniqueness of flows is the one using effectively acyclic flow graphs. A flow graph (N, e, f) is effectively acyclic iff. the flow along each cycle within (N, e, f) is 0:

```
definition eff_acyclic where
"eff_acyclic N e f ≡ ∀k ≥ 1. ∀ns ∈ k_lists N k. chain e ns (hd ns) (f (hd ns)) = 0"
```

Remember the definition of k-lists:

```
definition "k_lists  X k ≡ { xs. set xs ⊆ X ∧ length xs = k }"
```

We start with the proof of uniqueness. For a given effectively acyclic flow graph (N, e, f) with inflow i, flow f is unique:

```
lemma eff_acyclic_flow_unique:
  fixes E :: "('m ⇒ 'm :: pos_cancel_comm_monoid_add) set"
  assumes "End_closed E" "flow_eq (fg N e f) i" "eff_acyclic N e f"
    "finite N" "∀x y. e x y ∈ E" "(λ_. 0) ∈ E"
  shows "∀f'. flow_eq (fg N e f') i ⟶ eff_acyclic N e f' ⟶ f = f' on N"
```

Our formulation of this lemma is a little bit low-level as we have to explicitly state the shared inflow of two flow graphs (as it was already the case for lemma `nilpotent_flow_exists`).

Proof. Assume a second effectively acyclic flow graph (N, e, f') that satisfies the flow equation together with inflow i. We have to show for all $n \in N$ that $f\ n = f'\ n$.

Let $n \in N$. Let $l \coloneqq |N| + 1$. First, we note for all paths ns with $|ns| = l$ that due to $ns \subseteq N$ the pigeonhole principle provides us some $m \in ns$ that occurs at least twice in ns. Let $ns = ns_1 \cdot m \cdot ns_2 \cdot m \cdot ns_3$ be a decomposition of ns that exposes two occurences of m. By effective acyclicity of (N, e, f) and (N, e, f') we obtain

$$f\ m \triangleright \mathsf{chain}\ \ e\ (m \cdot ns_2)\ m = 0$$
$$f'\ m \triangleright \mathsf{chain}\ \ e\ (m \cdot ns_2)\ m = 0$$

As our edge functions are endomorphisms this result extends to chains over ns:

$$\begin{aligned}
&f\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\ \ e\ ns\ m \\
&= f\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ ns_1\ m \triangleright \mathsf{chain}\, e\ (m \cdot ns_2)\ m \triangleright \mathsf{chain}\, e\ (m \cdot ns_3)\ n \\
&= f\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ ns_1\ m \triangleright 0 \triangleright \mathsf{chain}\, e\ (m \cdot ns_3)\ n \\
&= 0 \triangleright \mathsf{chain}\, e\ (m \cdot ns_3)\ n \\
&= 0
\end{aligned}$$

Function 0 renders all arguments to 0, therefore, we can ignore its argument $f\ (\mathsf{hd}\ ns \triangleright \mathsf{chain}\, e\ ns_1\ m)$ and evaluate this term to 0. As $\mathsf{chain}\, e\ (m \cdot ns_3)\ n$ is an endomorphisms this value is also the final result. Using this and the analogous derivation for f', we can calculate (with implicit constraints $ns \subseteq N$):

$$\begin{aligned}
f\ n &= i\ n + (\sum_{|ns|<l} i\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ ns\ n) + (\sum_{|ns|=l} f\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ ns\ n) \\
&= i\ n + (\sum_{|ns|<l} i\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ ns\ n) \\
&= i\ n + (\sum_{|ns|<l} i\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ ns\ n) + (\sum_{|ns|=l} f'\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ ns\ n) \\
&= f'\ n
\end{aligned}$$

The first and last equality are justified by lemma `unroll_flow_eq'`. The other two equalities apply $f\ (\mathsf{hd}\ ns) \triangleright \mathsf{chain}\, e\ (m \cdot ns_2)\ m = 0$ for $|ns| = l$. ■

To prove the existence of flow in effectively acyclic flow graphs we need another kind of extension:

```
definition subflow_preserving_extension ("_ ≲S _") where
"h ≲S h' ≡ int_fg h ≲ int_fg h' ∧
(∀n ∈ dom_fg h. ∀n' ∈ -dom_fg h'. ∀m ≤ inf_fg h n. cap_fg h n n' m = cap_fg h' n n' m) ∧
(∀n ∈ dom_fg h' - dom_fg h. ∀n' ∈ -dom_fg h'. ∀m ≤ inf_fg h' n. cap_fg h' n n' m = 0)"
```

Subflow-preserving extensions extend contextual extensions but work on the level of flow graphs instead of flow interfaces only. The additional constraints introduced by the subflow-preserving extension are that: first, the original flow graph embedded into the extended flow graph must have the same capacities to the outside as the old flow graph, and second, the added portion in the extended flow graph must not have outflow at all.

The existence of flows in effectively acyclic flow graphs was supposed to be formalized by the following lemma:

```
lemma maintain_eff_acyclic_dom_flawed:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes nbot: "h = h1 + h2" "h ≠ bot" "h1' ≠ bot"
    and dom: "dom_fg h1' ∩ dom_fg h2 = {}"
    and edge: "∀n ∈ dom_fg h1' - dom_fg h1. outf_fg h2 n = 0"
        "∀x y. edge_fg h x y ∈ E" "End_closed E"
        "id ∈ E" "∀x y. edge_fg h2 x y ∈ E"
        "∀x y. edge_fg h1 x y ∈ E" "∀x y. edge_fg h1' x y ∈ E"
        "∀x y. edge_fg (h1' + h2) x y ∈ E" "(λ_. 0) ∈ E" "reduced E"
    and ea: "eff_acyclic' h" "eff_acyclic' h1" "eff_acyclic' h1'"
      "eff_acyclic' h2" "h1 ≲S h1'"
    shows "∃h'. h' = h1' + h2 ∧ h' ≠ bot ∧ eff_acyclic' h' ∧ h ≲S h'"
```

Unfortunately, `maintain_eff_acyclic_dom_flawed` turned out to be unprovable (see counter-example later) because of an flawed instantiation of `reduced E`. Instead we will prove a fixed variant `maintain_eff_acyclic_dom` using the interpretation of reduced that reflects the flawed instantiation:

```
definition pointwise_reduced where
  "pointwise_reduced E ≡ ∀e ∈ E. ∀x. e x ≠ 0 ⟶ e (e x) ≠ 0"
```

Definition `pointwise_reduced` is still sufficient to prove at least some of the examples in [Krishna et al., 2020], e.g. path counting flows.
Theorem `maintain_eff_acyclic_dom` will be proven separately in Chapter 6 due to the proof's significant size and required development. Here, we provide a counter-example for `maintain_eff_acyclic_dom_flawed`. The insight of the counter-example is that the subflow-preserving extension does not prevent multi-dimensional flow-domains like multisets to close cycles. We are able to close a cycle within a subgraph by hiding the effects that would contradict the subflow-preserving extension just before leaving the subgraph. Consider an operator f_R on multisets

$$f_R\ X = \{x + 1 \mid x \in X \wedge x \notin R\}$$

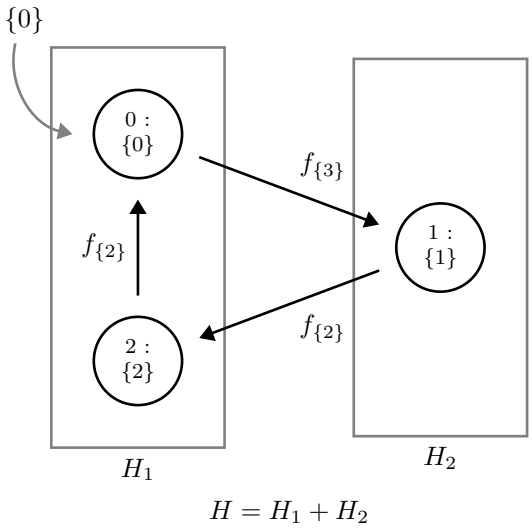
that first filters its argument X for all elements in R and then increments the remaining values. In our formalization we show that the closure E over all functions f_R and $\lambda\_.\ 0$ and id is a reduced set of endomorphisms. Figure 4.1a shows us a simple effectively acyclic flow graph $H = H_1 + H_2$:

1. H is a flow graph because its domain is finite and for $f = \{0 \mapsto \{0\}, 1 \mapsto \{1\}, 2 \mapsto \{2\}\}$ and $i = \{0 \mapsto \{0\}, \_ \mapsto \emptyset\}$ the flow equation holds: $\{0\} = \{0\} + (\emptyset + \emptyset + f_{\{2\}}\ \{2\})$, $\{1\} = \emptyset + (f_{\{3\}}\ \{0\} + \emptyset + \emptyset)$, and $\{2\} = \emptyset + (\emptyset + f_{\{2\}}\ \{1\} + \emptyset)$.

2. H is effectively acyclic because for each cycle $(0,1,2)$, $(1,2,0)$, $(2,0,1)$ not immediately containing a 0-edge it holds that $\{0\} \triangleright f_{\{3\}} \triangleright f_{\{2\}} \triangleright f_{\{2\}} = \emptyset$, $\{1\} \triangleright f_{\{2\}} \triangleright f_{\{2\}} \triangleright f_{\{3\}} = \emptyset$, and $\{2\} \triangleright f_{\{2\}} \triangleright f_{\{3\}} \triangleright f_{\{2\}} = \emptyset$.

$H = H_1 + H_2$ being an effectively acyclic flow graph implies that H_1 and H_2 are effectively acyclic flow graphs, too. H_1' is also an effectively acyclic graph (inflow $i = \{0 \mapsto \{0\}, 2 \mapsto \{2\}\}$, obviously no cycles without zero-edges). Furthermore, $H_1' \cap H_2 = \emptyset$. The condition that H_2 must not have outflow to $\mathsf{dom}\ H_1' - \mathsf{dom}\ H_1 = \emptyset$ is trivially true. Our edge functions are reduced and contained in $E \supseteq \{(\lambda\_.\ 0), \mathrm{id}\}$ (see formalization of counter-example). It remains to show that $H_1 \preccurlyeq_S H_1'$:

- $\mathsf{int}\ H_1 \preccurlyeq \mathsf{int}\ H_1'$ because of $\mathsf{dom}\ H_1 = \mathsf{dom}\ H_1'$, $\mathsf{inf}\ H_1 = \{0 \mapsto \{0\}, 2 \mapsto \{2\}\} = \mathsf{inf}\ H_1'$, and $\mathsf{outf}\ H_1 = \{1 \mapsto \{1\}\} = \mathsf{outf}\ H_1'$.



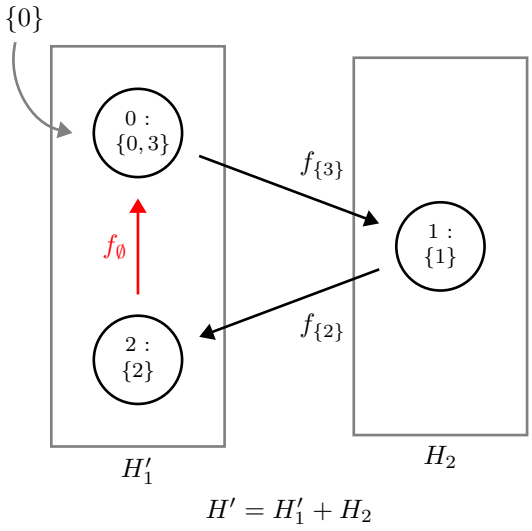


Figure 4.1.: Counter-example for theorem 3.38. Gray boxes indicate subgraphs, cycles represent nodes with labels stating their identifier and flow, black arrows represent non-0-edges annotated with their edge function, gray arrows represent inflow.

- $\forall n \in H_1, n' \in -H_1', m \leq \mathsf{inf}\ H_1\ n.\ \mathsf{cap}\ H_1\ n\ n'\ m = \mathsf{cap}\ H_1'\ n\ n'\ m$ because of

$$\mathsf{cap}\ H_1\ 0\ 1\ \emptyset = f_{\{3\}}\ \emptyset = m = f_{\{3\}}\ \emptyset = \mathsf{cap}\ H_1'\ 0\ 1\ \emptyset$$
$$\mathsf{cap}\ H_1\ 0\ 1\ \{0\} = f_{\{3\}}\ \{0\} = \{1\} = f_{\{3\}}\ \{0\} = \mathsf{cap}\ H_1'\ 0\ 1\ \{0\}$$

  and

$$\mathsf{cap}\ H_1\ 2\ 1\ \emptyset = f_{\{3\}}\ (f_{\{2\}}\ \emptyset) = \emptyset = f_{\{3\}}\ (f_\emptyset\ \emptyset) = \mathsf{cap}\ H_1'\ 2\ 1\ \emptyset$$
$$\mathsf{cap}\ H_1\ 2\ 1\ \{2\} = f_{\{3\}}\ (f_{\{2\}}\ \{2\}) = \emptyset = f_{\{3\}}\ (f_\emptyset\ \{2\}) = \mathsf{cap}\ H_1'\ 2\ 1\ \{2\}$$

- $\forall n \in H_1' - H_1, n' \in -H_1', m \leq \mathsf{inf}\, H_1'.\ m \triangleright \mathsf{cap}\, H_1'\ n\ n'\ m = 0$ holds trivially due to $\mathsf{dom}\, H_1 = \mathsf{dom}\, H_1'$.

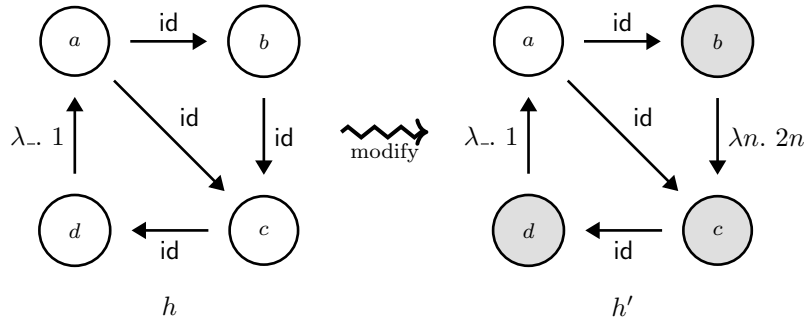
According to `maintain_eff_acyclic_dom_flawed` $H' = H_1' + H_2$ should be effectively acyclic. However, in Figure 4.1b there is cycle $(0, 1, 2)$ in $H' = H_1' + H_2$ such that

$$\{0\} \triangleright f_{\{3\}} \triangleright f_{\{2\}} \triangleright f_{\emptyset} = \{3\} \neq \emptyset$$

Hence, H' is not effectively acyclic in contradiction to the lemma.

4.3. Flow Footprints

This section presents a theoretical result about footprints of modifications to flow graphs. Given a flow graph h, when we apply an operation on h and obtain a result flow graph h' we call the sub-flow graph $h_1' \subseteq h'$ that contains exactly the modified portions of h the *footprint* of the operation. The unmodified portions can be considered as the *frame* of the operation following Separation Logic's narrative. Therefore, the portion h_1 of a flow graph to be modified, the modified portion h_1' and the frame h_2 satisfy $h = h_1 + h_2$ and $h' = h_1' + h_2$. Consider for example flow graph h in the left where we replace edge (b, c) by $\lambda n.\ 2n$:



In this example we have for inflows $i = i' = \{a \mapsto 1, \_ \mapsto 0\}$ and flows $f = \{a \mapsto 2, b \mapsto 2, c \mapsto 4, d \mapsto 4\}$ and $f' = \{a \mapsto 2, b \mapsto 2, c \mapsto 6, d \mapsto 6\}$ that $\mathsf{dom}\, h_1 = \mathsf{dom}\, h_1' = \{b, c, d\}$ and $\mathsf{dom}\, h_2 = \{a\}$. b is part of the footprint because one of its outgoing edge functions changed, c and d are part of the footprint because their flow changed. The constant edge function from d to a contains the propagation of the effects of our modification such that a remains unaffected.

The crucial insight that lemma `in_footprint_iff_changed` will provide us about flow footprints is that the interfaces of h_1 and h_1' are identical, i.e. $\mathsf{int}\, h_1 = \mathsf{int}\, h_1'$. Therefore, we are able to determine suitable flow interfaces for our operations by looking at their effect.

Formally, we define footprints as [Krishna et al., 2019, def. 7]:

```
definition flow_footprint where
  "flow_footprint ≡ λh h' h1'.
            (∃h1 h2. h = h1 + h2 ∧ h' = h1'  + h2 ∧ int_fg h1 = int_fg h1' ∧
     (∀h1''. (∃h1 h2. h = h1 + h2 ∧ h' = h1'' + h2 ∧ int_fg h1 = int_fg h1'')
              ⟶ (dom_fg h1' ⊆ dom_fg h1'')))"
```

This definition seems a little convoluted: it states for an original flow graph h, an modified flow graph h' and a flow graph h_1' if h_1' is the flow footprint of the modification that transformed h into h'. Flow graph h is decomposed into $h = h_1 + h_2$ and h' into $h' = h_1' + h_2$. As h_1' is supposed the smallest flow graph that satisfies this predicate flow graph h_2 is supposed to be the largest

flow graph. Therefore, all nodes of h' that were not changed by the operation must be part of h_2 and h'_1 comprises exactly the changed nodes.

Our intuition from above is justified by the following lemma: the flow footprint of an original flow graph h and a modified flow graph h' is exactly the flow graph h'_1 that contains all nodes in h' that were changed by the modification of h to h' [Krishna et al., 2019, lem. 4].

```
lemma in_footprint_iff_changed:
  assumes "int_fg h = int_fg h'" "h ≠ bot" "h' ≠ bot"
    "n ∈ dom_fg h" "flow_footprint h h' h1'"
  shows "n ∈ dom_fg h1' ⟷
    fg {n} (edge_fg h) (flow_fg h) ≠ fg {n} (edge_fg h') (flow_fg h')"
```

Proof. We prove the first direction by contradiction by assuming $n \in h'_1$ and $(\{n\}, e, f) = (\{n\}, e', f')$. In this case we would be able to shrink h'_1 to $h'_1 \setminus \{n\}$ and therefore would violate the minimality of h'_1. Therefore, we have to construct flow graphs g''_1, g_1 and g_2 such that $h = g_1 + g_2$, $h' = g''_1 + g_2$, $\mathsf{int}\, g''_1 = \mathsf{int}\, g_1$, and $\mathsf{dom}\, g''_1 \subsetneq \mathsf{dom}\, h'_1$.

By assumption we obtain h_1 and h_2 such that h'_1 is minimal, $h = h_1 + h_2$, $h' = h'_1 + h_2$ and $\mathsf{int}\, h_1 = \mathsf{int}\, h'_1$. From the last fact follows that $\mathsf{dom}\, h_1 = \mathsf{dom}\, h'_1$.

We notice that $\mathsf{dom}\, h = (\mathsf{dom}\, h_1 \setminus \{n\}) \dot\cup (\mathsf{dom}\, h_2 \cup \{n\})$ due to $h = h_1 + h_2$, $n \in h'_1$ and $\mathsf{dom}\, h_1 = \mathsf{dom}\, h'_1$. An application of `split_fg` then provides us

$$h = (\mathsf{dom}\, h_1 \setminus \{n\}, e, f) + (\mathsf{dom}\, h_2 \cup \{n\}, e, f) \neq \bot$$

A second application of `split_fg` provides us

$$h' = (\mathsf{dom}\, h_1 \setminus \{n\}, e', f') + (\mathsf{dom}\, h_2 \cup \{n\}, e', f') \neq \bot$$

using $\mathsf{dom}\, h'_1 = \mathsf{dom}\, h_1$. Due to $h' = h'_1 + h_2$ and $h = h_1 + h_2$ we know that $f' = f$ on $\mathsf{dom}\, h_2$ and $e' = e$ on $\mathsf{dom}\, h_2$. Moreover, $(\{n\}, e, f) = (\{n\}, e', f')$, i.e. $e\ n = e'\ n$ and $f\ n = f'\ n$. This combines to $f' = f$ on $\mathsf{dom}\, h_2 \cup \{n\}$ and $e' = e$ on $\mathsf{dom}\, h_2 \cup \{n\}$. Therefore, we can rewrite the previous representation of h' to

$$h' = (\mathsf{dom}\, h_1 \setminus \{n\}, e', f') + (\mathsf{dom}\, h_2 \cup \{n\}, e, f)$$

We then derive using `int_fg_fi_hom` and $h' \neq \bot$ that

$$\begin{aligned} &\mathsf{int}\, (\mathsf{dom}\, h_1 \setminus \{n\}, e, f) + \mathsf{int}\, (\mathsf{dom}\, h_2 \cup \{n\}, e, f) \\ &= \mathsf{int}\, h = \mathsf{int}\, h' \\ &= \mathsf{int}\, (\mathsf{dom}\, h_1 \setminus \{n\}, e', f') + \mathsf{int}\, (\mathsf{dom}\, h_2 \cup \{n\}, e, f) \end{aligned}$$

Using cancellativity of flow interfaces we conclude $\mathsf{int}\, (\mathsf{dom}\, h_1 \setminus \{n\}, e, f) = \mathsf{int}\, (\mathsf{dom}\, h_1 \setminus \{n\}, e', f')$. Therefore, we constructed some flow graphs $g''_1 \coloneqq (\mathsf{dom}\, h_1 \setminus \{n\}, e', f')$, $g_1 \coloneqq (\mathsf{dom}\, h_1 \setminus \{n\}, e, f)$ and $g_2 \coloneqq (\mathsf{dom}\, h_2 \cup \{n\}, e, f)$ such that $h = g_1 + g_2$, $h' = g''_1 + g_2$, $\mathsf{int}\, g''_1 = \mathsf{int}\, g_1$, and $\mathsf{dom}\, g''_1 = \mathsf{dom}\, h_1 \setminus \{n\} \subsetneq \mathsf{dom}\, h_1 = \mathsf{dom}\, h'_1$. This contradicts our minimality assumption regarding h'_1 from `flow_footprint` $h\ h'\ h'_1$.

The second direction provides us the assumption that $(\{n\}, e, f) \neq (\{n\}, e', f')$ and we have to show that $n \in \mathsf{dom}\, h'_1$. By assumption we obtain some flow graphs h_1 and h_2 such that h'_1 is minimal, $h = h_1 + h_2$, $h' = h'_1 + h_2$ and $\mathsf{int}\, h_1 = \mathsf{int}\, h'_1$.

We observe that $(\{n\}, e, f) \neq (\{n\}, e', f')$ only can hold if $e\ n \neq e'\ n$ or $f\ n \neq f'\ n$. From this we can conclude that $n \in \mathsf{dom}\, h_1$ because for $n \in \mathsf{dom}\, h_2$ we have due to $h = h_1 + h_2$ and $h' = h'_1 + h_2$ that $e\ n = e'\ n$ and $f\ n = f'\ n$ and this contradicts our assumption that $e\ n \neq e'\ n$ or $f\ n \neq f'\ n$.

■

We can show that flow footprints are unique:

```
lemma flow_footprint_unique:
  assumes "flow_footprint h h' h1'" "flow_footprint h h' h1''" "h ≠ bot" "h' ≠ bot"
  shows "h1' = h1''"
```

Proof. By assumption there are flow graphs $h_{1,1}$ and $h_{1,2}$ such that $h = h_{1,1} + h_{1,2}$, $h' = h'_1 + h_{1,2}$ and $\mathsf{int}\, h_{1,1} = \mathsf{int}\, h'_1$, as well as flow graphs $h_{2,1}$ and $h_{2,2}$ such that $h = h_{2,1} + h_{2,2}$, $h' = h''_1 + h_{2,2}$ and $\mathsf{int}\, h_{2,1} = \mathsf{int}\, h''_1$.

We can derive using `int_fg_fi_hom` that

$$\bot \neq \mathsf{int}\, h = \mathsf{int}\,(h_{1,1} + h_{1,2}) = \mathsf{int}\, h_{1,1} + \mathsf{int}\, h_{1,2} = \mathsf{int}\, h'_1 + \mathsf{int}\, h_{1,2} = \mathsf{int}\,(h'_1 + h_{1,2}) = \mathsf{int}\, h'$$

First, we prove that $h_{1,2} = h_{2,2}$. To prove $h_{1,2} = h_{2,2}$ we have to show that $\mathsf{dom}\, h_{1,2} = \mathsf{dom}\, h_{2,2}$. For this we define a set

$$X = \{n \mid n \in \mathsf{dom}\, h \wedge (\{n\}, e, f) \neq (\{n\}, e', f')\}$$

i.e. the set of nodes on which h and h' differ. We now show that $X = \mathsf{dom}\, h'_1$. To show $X \subseteq \mathsf{dom}\, h'_1$ let $n \in X$ and apply the second direction of `in_footprint_iff_changed`: we already showed $\mathsf{int}\, h = \mathsf{int}\, h'$; $h, h' \neq \bot$ holds by assumption; $n \in \mathsf{dom}\, h$ because of the definition of X; `flow_footprint h h' h'1` holds by assumption; the inequality between the singleton flow-graphs holds by definition of X. We obtain $n \in \mathsf{dom}\, h'_1$. To show $\mathsf{dom}\, h'_1 \subseteq X$ let $n \in \mathsf{dom}\, h'_1$. Note that

$$n \in \mathsf{dom}\, h'_1 \subseteq \mathsf{dom}\,(h'_1 + h_{1,2}) = \mathsf{dom}\,(\mathsf{int}\,(h'_1 + h_{1,2})) = \mathsf{dom}\,(\mathsf{int}\,(h_{1,1} + h_{1,2})) = \mathsf{dom}\, h \quad (4.5)$$

In this derivation $\mathsf{dom}\,(\mathsf{int}\,(h'_1 + h_{1,2})) = \mathsf{dom}\,(\mathsf{int}\,(h_{1,1} + h_{1,2}))$ followed from assumption $\mathsf{int}\, h_{1,1} = \mathsf{int}\, h'_1$ and `int_fg_fi_hom`. Equation (4.5) immediately proves our first subgoal $n \in \mathsf{dom}\, h$. Furthermore, we show $(\{n\}, e, f) \neq (\{n\}, e', f')$ using the first direction of `in_footprint_iff_changed`: we already showed $\mathsf{int}\, h = \mathsf{int}\, h'$; $h, h' \neq \bot$ holds by assumption; $n \in \mathsf{dom}\, h$ by Equation (4.5); `flow_footprint h h' h'1` still holds by assumption; $n \in \mathsf{dom}\, h'_1$ by assumption.

Analogously, we can show $X = \mathsf{dom}\, h_{2,1}$ and obtain $\mathsf{dom}\, h_{1,1} = \mathsf{dom}\, h_{2,1}$. Then, $h_{1,1} + h_{1,2} = h = h_{2,1} + h_{2,2}$ implies $\mathsf{dom}\, h_{1,2} = \mathsf{dom}\, h_{2,2}$. We can conclude that $h_{1,2} = (\mathsf{dom}\, h_{1,2}, e, f) = (\mathsf{dom}\, h_{2,2}, e, f) = h_{2,2}$. Using this we can conclude $h'_1 + h_{1,2} = h' = h''_1 + h_{2,2} = h''_1 + h_{1,2}$. Cancellativity of flow graphs then provides us $h'_1 = h''_1$. ■

Considering the previous lemma we can justify the following definition:

```
definition "the_flow_footprint ≡ λh h'. (THE h1'. flow_footprint h h' h1')"
```

This definition concludes our formalization of flow interfaces and graphs. The next chapter embeds them into a Separation Logic framework and applies the Flow Framework to an example.

5. Application of the Flow Framework

This chapter first formalizes the Flow Framework as presented in [Krishna et al., 2020, sec. 4.1] which allows us to embed reasoning using flow graphs and flow interfaces into Isabelle's existing Separation Logic framework for Imperative-HOL [Lammich and Meis, 2012]. Then we demonstrate the application of the Flow Framework on the example of a graph update algorithm presented in [Krishna et al., 2020]. The example is inspired by the bookkeeping algorithm of the priority inheritance protocol.

5.1. The Flow Framework

We encapsulate the Flow Framework in an Isabelle locale such that we can instantiate the framework for different use cases and obtain a common infrastructure for all use cases:

```
locale FF =
  fixes γ :: "'a ⇒ 'n ref ⇒ 'n :: heap ⇒ 'm :: cancel_comm_monoid_add ⇒ bool"
    and φ :: "('n ref, 'm) fi ⇒ bool"
    and edges :: "'n ref ⇒ 'n ⇒ 'n ref ⇒ 'm ⇒ 'm"
  assumes edges_not_refl: "⋀x fs g m. γ g x fs m ⟹  edges x fs x = (λ_. 0)"
begin
```

The parameterization of our locale consists of a node type ‘n, a flow domain ‘m, a custom type ‘a, the node-local invariant γ, the global invariant φ and the function `edges`. Our variant of the Flow Framework assumes that `edges` does not generate reflexive edges as this implies that for each node a singleton flow graph exists. Applications of the Flow Framework are supposed to be encoded into this format.

5.1.1. Flow Graphs

The basic idea of the Flow Framework is to abstract heap objects into nodes `N g x fs m h`:

```
definition N :: "'a ⇒ 'n ref ⇒ 'n ⇒ 'm ⇒ ('n ref, 'm) fg ⇒ assn" where
"N g x fs m h ≡
  x ↦ᵣ fs * ↑(γ g x fs m ∧ h ≠ bot ∧ h = fg {x} (λ_. edges x fs) (λ_. m))"
```

Argument `x` is the heap location of the heap object abstracted by a node. Argument `fs` exposes the content of this particular heap location to the outside of the predicate, i.e a Flow Framework node encapsulates a Separation Logic heap object `x` $\mapsto$ `fs`. Note that we identify nodes using their heap locations. Argument `m` allows us to specify the flow in nodes. In principle the choice of this argument is arbitrary, however, it must suit the proof we are conducting. Argument `h` exposes the singleton flow graph induced by the abstracted heap object and the provided flow `m`. We see in the definition of `N` that the singleton flow graph for our node `x` is always generated the same way: the explicitly specified flow argument `m` becomes the flow in `x` and `edges` generates the edge functions from `x` to other nodes. These edges are generated from the content of the abstracted heap object, i.e. the flow graph is induced by the underlying heap objects. Therefore, we do not specify flow graphs explicitly but they are defined implicitly by the abstracted heap

objects. Hence, changes to heap objects also affect the structure of related flow graphs. The great challenge in the Flow Framework is to ensure that changes do not invalidate the flow in a flow graph. All development in Section 4.2 serves this cause. Our framework enforces flow graph validity checks by only allowing modifications to separated nodes that must be reintegrated with the remainder of their data structure and reintegration requires a valid composite flow graph. The singleton flow graph induced by a node is always valid due to our locale assumption `edges_not_refl`. Additionally each node ensures that its flow satisfies the node-local invariant γ. γ enables us to impose constraints on the flow a node can assume. Finally, `N` has some argument g that allows the user to inject auxiliary information into `N`. `g` can be related to the heap content and flow using γ.

We continue with the definition of graphs, i.e. compositions of multiple nodes:

definition
```
  Gr :: "('n ref ⇒ 'a) ⇒ ('n ref ⇒ 'n) ⇒ 'n ref set ⇒ ('n ref, 'm) fg ⇒ assn"
where
  "Gr η Fs X h ≡
    ∃A H M. (∏x ∈ X. N (η x) x (Fs x) (M x) (H x)) *
          ↑(finite X ∧ h ≠ bot ∧ h = (∑x ∈ X. H x))"
```

`X` is a set of heap locations. A graph assumes that each of these heap locations represents a node. The related heap content is point-wisely accumulated and exposed by a function `Fs`. Similarly, function η point-wisely accumulates and exposes all auxiliary information `g` of all nodes within the graph. We also accumulate the flow arguments `m` and flow graphs `h` of all nodes of the graph using existentially quantified variables `M` and `H`. `H` is used to declare the flow graph `h` of graphs which is exactly the sum of all singleton flow graphs `H x` of nodes `x` ∈ `X`. `M` simply captures the flow given to each node in the graph. We do not use this information in our examples but it could be exposed similarly to `Fs` or γ if needed. While `h` ≠ `bot` is only stated for convenience in the definition of nodes `N` this constraint is more important for graphs `Gr`: the composition of all flow graphs `H x` might indeed be invalid. To ensure that the data structure represented by a graph always maintains a valid invariant we have to enforce $\mathtt{h} \neq \bot$. Nodes can be added to a graph only if the composition of the graph and the new nodes is valid. To apply changes to parts of the graph those parts must be removed first and reintegrated after the change. This way we maintain a valid flow graph.

Now we derive some useful entailments to reason about graphs and nodes during verification. The first entailment is the decomposition entailment. It enables us to split a flow graph `h` into two flow graphs $\mathtt{h}_1$ and $\mathtt{h}_2$ such that $\mathtt{h} = \mathtt{h}_1 + \mathtt{h}_2$:

lemma `decomp_Gr`:
```
  assumes "xs1 ∩ xs2 = {}" "dom_fg h = xs1 ∪ xs2"
  shows "Gr η Fs (xs1 ∪ xs2) h ⟹A
    ∃A h1 h2. Gr η Fs xs1 h1 * Gr η Fs xs2 h2 * ↑(h ≠ bot ∧ h = h1 + h2)"
```

This entailment applies `split_fg`. The corresponding inverse entailment is the composition entailment. It allows us to combine two flow graphs h_1 and h_2 into a single flow graph $h_1 + h_2$:

lemma `comp_Gr`:
```
  assumes "xs1 ∩ xs2 = {}" "finite xs1" "finite xs2" "h = h1 + h2" "h ≠ bot"
  shows "Gr η1 Fs1 xs1 h1 * Gr η2 Fs2 xs2 h2 ⟹A
    Gr (λx. if x ∈ xs1 then η1 x else η2 x)
       (λx. if x ∈ xs1 then Fs1 x else Fs2 x) (xs1 ∪ xs2) h"
```

Precondition $\mathtt{h}_1 + \mathtt{h}_2 \neq \bot$ ensures that we obtain a valid flow in the composed graph as required by the definition of `Gr` and already discussed.

To construct and deconstruct singleton graphs from nodes we have two entailments:

```
lemma sing_N_Gr: "N (η x) x fs m h ⟹A Gr η (λ_. fs) {x} h"

lemma sing_Gr_N:
  "Gr η Fs {x} h ⟹A ∃Am. N (η x) x (Fs x) m h *
    ↑(γ (η x) x (Fs x) m ∧ h ≠ bot ∧ h = fg {x} (λ_. edges x (Fs x)) (λ_. m))"
```

In entailment `sing_Gr_N` we already provide all information that is available on the node because it might be relevant for some verification steps without actually modifying the underlying heap objects. For example the "fake-interface" approach that we will use later only manipulates the interface of some nodes without modifying the underlying heap object. In order to do so we need the node's flow graph.

For additional convenience we combine the above four entailments into two entailments "fold" and "unfold" that directly separate single nodes from graphs and reinsert single node into graphs:

```
lemma unfold_N:
  assumes "x ∈ X" "X = dom_fg h"
  shows "Gr η Fs X h ⟹A
    ∃Am h1 h'. Gr η Fs (X - {x}) h' * N (η x) x (Fs x) m h1 *
      ↑(h = h1 + h' ∧ γ (η x) x (Fs x) m ∧ h1 = fg {x} (λ_. edges x (Fs x)) (λ_. m))"

lemma fold_N:
  assumes "x ∉ X" "X = dom_fg h" "h + h1 ≠ bot" "γ g x fs m"
  shows "Gr η Fs X h * N g x fs m h1 ⟹A
    Gr (λy. if y = x then g else η y) (λy. if y = x then fs else Fs y) ({x} ∪ X) (h1 + h)"
```

To obtain access to the encapsulated heap objects we provide two entailments to "open" and "close" nodes (the terms "fold" and "unfold" that are usually used in the context of definitions are already taken):

```
lemma open_N:
  shows "N g x fs m h ⟹A
    x ↦r fs * ↑(h ≠ bot ∧ γ g x fs m ∧ h = fg {x} (λ_. edges x fs) (λ_. m))"

lemma close_N:
  assumes "γ g x fs m"
  shows "x ↦r fs ⟹A N g x fs m (fg {x} (λ_. edges x fs) (λ_. m))"
```

Opening a node provides us the underlying heap object and all information available about the node, i.e. its local invariant and singleton flow graph. Closing a node requires us to state a flow `m` and auxiliary information `g`, and to prove that the local invariant holds for the provided heap object `fs` and `g` and `m`. `close_N` also nicely demonstrates how singleton flow graphs are induced by heap objects and explicitly stated flow `m`.

There are more entailments presented in [Krishna et al., 2020], e.g. an entailment to invoke the replacement theorem. Please refer to our formalization for those entailments.

If we remember our intuition from Section 2.3 we said that the Flow Framework manages an unstructured set of heap locations in order to obtain the flexibility that is required to describe data structures with an inherent degree of sharing. The definition of graphs and related operations exactly exhibit this intuition: we have the set `X` of heap locations that can be arbitrarily partitioned into smaller flow graphs and joined into larger flow graphs. Unfolding and opening enables us to construct excerpts of data structures in order to apply modifications on the underlying heap objects. The iterated separating conjunction at the heart of the definition

of graphs enables these operations on graphs as we can arbitrarily partition and combine the iterated separation conjunction.

Furthermore, we said that we needed an approach flexible enough to state invariants in this context. Flow graphs and flow interfaces provide this flexibility: they are separation algebras and can be split arbitrarily to suit the construction of excerpts of data structures. The price for this flexibility is that we are constrained in the modifications that can be applied: we must preserve flow interfaces (or more generally prove the existence of flows) such that we are able to recompose the excerpts of data structures with their remainders.

5.1.2. Flow Interfaces

So far, we only addressed abstractions for flow graphs. To obtain the greater flexibility of working with flow interfaces instead of flow graphs we lift the development of the entire previous subsection to the flow interface level. Interface nodes of the Flow Framework simply encapsulate the original nodes and expose the flow interface of the encapsulated flow graph instead of the flow graph itself:

```
definition NI :: "'a ⇒ 'n ref ⇒ 'n ⇒ 'm ⇒ ('n ref, 'm) fi ⇒ assn" where
"NI g x fs m i ≡ ∃A h. N g x fs m h * ↑(i ≠ bot ∧ i = int_fg h)"
```

Analogously, we also lift the definition of graphs of the Flow Framework by exposing the flow interface of the flow graph of the encapsulated graph:

```
definition
  GrI :: "('n ref ⇒ 'a) ⇒ ('n ref ⇒ 'n) ⇒ 'n ref set ⇒ ('n ref, 'm) fi ⇒ assn"
where
  "GrI η Fs X i ≡ ∃A h. Gr η Fs X h * ↑(i = int_fg h)"
```

As flow interfaces are sufficient to determine composability of graphs we simplify composition and furthermore obtain the ability to substitute flow graphs for equivalent ones.

There are entailments completely analogous to the ones of the previous subsection. We omit these entailments here and notice that their naming scheme simply substituted `N` and `Gr` by `NI` and `GrI`, e.g. `unfold_NI` instead of `unfold_N`.

5.1.3. Proof Rules

More interesting is how to apply these entailments during verification. Often, quite precise application of our entailments is required. In order to obtain some degree of precision we introduce ghost commands for Imperative-HOL and corresponding Separation Logic proof rules that "execute" these ghost commands during verification. For example to fold a node into a graph using `fold_GrI` we introduce the ghost command

```
definition "fold_GrI x ≡ return ()"
```

Obviously, during execution this command has no effect. However, during verification we can trigger effects (on the logic level) using this command: in the case of the `fold_GrI` ghost command we want rule `fold_GrI` to be applied on the current proof state:

```
lemma fold_GrI_rule:
  assumes "i1 + i2 ≠ bot" "x ∉ X" "finite X"
  shows "<NI g x fs m i1 * GrI η Fs X i2>
    fold_GrI x
    <λ_. GrI (λy. if y ∈ {x} then g else η y)
             (λy. if y ∈ {x} then fs else Fs y) ({x} ∪ X) (i1 + i2)>"
```

The proof of this proof rule on the level of Separation Logic level immediately reduces to our entailment. The argument of the ghost command specifies which node is supposed be inserted into an implicitly determined graph. The naming scheme of our proof rules is to simply append the suffix `_rule` to the names of the entailments.

We obtain some degree of precision and expressiveness by introducing proof rules like rule `fold_GrI_rule`. Precision in terms of being able to specify within our Imperative-HOL programs when to apply which proof rule. This facility greatly simplifies proofs using the Flow Framework on its current low level of automation. Expressiveness is unfortunately limited as we currently can not specify into which graph the specified node is supposed to be inserted. Therefore, this proof rule is only suited for use cases that only exhibit a single graph (like the ones we will encounter later). The issue we have with graphs is that in contrast to nodes we are not able to identify graphs: ghost commands are stated within the program definition in Imperative-HOL, i.e. we do not have access to logical variables from this context and in general domains of graphs are logical variables (see for example Figure 5.4, there is no set `xs` corresponding to the graph that is later used in `update_correct`). In contrast, nodes fortunately already can be identified within the context of Imperative-HOL program definitions by the corresponding heap locations of heap objects that are required on the program level anyways. To overcome the limitation regarding the identification of graphs we could introduce ghost arguments in program definitions that represent domains of graphs that will later be introduced and provided to the program by proofs on these programs. During execution the ghost arguments of a program can be set to `undefined`. Of course, this approach requires some discipline to adhere to the convention that ghost arguments must not be occur in program level commands but only in ghost commands.

5.2. The PIP Example

5.2.1. The Priority Inheritance Protocol

The basic priority inheritance protocol [Sha et al., 1990], short PIP, is a job scheduling protocol where each job is assigned a priority and these priorities influence scheduling decisions if multiple jobs compete for resources: the higher the priority of a job the earlier it is supposed to gain access to resources and the earlier and/or longer it is supposed to be executed.

The issue of priority based job scheduling that the PIP is able to avoid is the following one: if a high priority job waits for some resource that currently is held by a low priority job then the high priority job has to wait for the low priority job to release the resource. Usually, waiting for the resource to become available again can not be circumvented. However, by waiting on the low priority job low priority scheduling decisions propagate to the execution of the high priority job as the low priority job is less privileged in terms of execution time or receipt of further required resources: a job of mid priority that blocks our low priority job also blocks the high priority job transitively. This behavior, called *priority inversion*, of course is undesirable. To prevent this behavior [Sha et al., 1990] introduce multiple *priority inheritance protocols*. The idea behind these protocols is that jobs that block jobs of higher priority "inherit" the priorities of the blocked jobs for the duration of their execution. Therefore, the progress of high priority jobs is no longer affected by low priority scheduling decisions because the progress of the blocking job is accelerated to the pace that the blocked jobs are at least entitled to.

5.2.2. The Example

In the following we will verify a potential data structure and algorithm for the priority and dependency bookkeeping of the "basic protocol" presented in [Krishna et al., 2020, ch. 3]. The

```
datatype fields = Fields "fields ref option" "int" "int" "int multiset"

fun γ :: "fields ref option ⇒ fields ref ⇒ fields ⇒ int multiset ⇒ bool" where
"γ η x (Fields y q q0 qs) m = (q0 ≥ 0 ∧ (∀q'∈set_mset qs. q' ≥ 0) ∧
                finite (set_mset qs) ∧
                m = qs ∧ Max'_mset ({#q0#} + qs) = q ∧
                η = y ∧ η ≠ Some x)"

definition φ where
"φ i ≡ i ≠ bot ∧ inf_fi i = (λ_. {#}) ∧ outf_fi i = (λ_. {#})"

fun edge :: "fields ref ⇒ fields ⇒ fields ref ⇒ (int multiset ⇒ int multiset)"
where
"edge x (Fields y q q0 qs) z =
  (λm. if Some z = y then {# Max_mset (m + {# q0 #}) #} else {#})"
```

Figure 5.1.: Instantiation of the basic Flow Framework locale for the PIP example

data structure is supposed to keep track of all jobs and resources and their relationship. In particular, we represent resources and jobs as nodes of a graph. The semantics of a directed edge depends on whether it points from a resource to a job or vice versa: an edge from a resource to a job indicates that the origin resource is held by the destination job, and an edge from a job to a resource indicates that the origin job wants to acquire the destination resource. Each job node of the graph is assigned a default priority. Additionally, each job node is assigned its "current" or "effective" priority that corresponds to the maximum of its own and all its predecessors' default priorities and controls scheduling decisions. When an edge is modified then the priorities of all successor nodes must possibly be adjusted: if an edge is added to the graph then the priority of the source node of the edge must be propagated to all successor nodes (at least as long as the priorities change), if an edge is removed from the graph then the default priority of the source node of the edge must be removed from all successor nodes.

Analogously to [Krishna et al., 2020] we slightly simplify the example by dropping the differentiation between resource nodes and job nodes, i.e. we only consider nodes. Each node is assigned a default priority, its current priority, a multiset of predecessor priorities, and of course its successor node (a single successor node is sufficient: resources can be locked by a single job only, jobs can wait for a single resource only).

We now have a look at our example implementation in Figures 5.1 to 5.4. First we introduce the data structure in Figure 5.1. We represent nodes as values of type `fields`. Each value of this type may contain a "pointer" to its potential successor node (accessible via an accessor function `fields_y`), its current priority (`fields_q`), its default priority (`fields_q0`), and some multiset of predecessor priorities (`fields_qs`). The data structure can be manipulated by two operations: `acquire` and `release`. `acquire` (Figure 5.2) adds an edge between two nodes and updates the current priorities of all nodes within the graph using `update` (Figure 5.4). Note that the code snippets already contain ghost commands which can be identified by the prefix '`ff`' for our flow framework instantiation. The direction of the edge that is added by `acquire` depends on whether the desired destination node is available (it does not have a successor) or not. If the desired destination node `r` is available then we insert edge (`p`, `r`), otherwise we add edge (`r`, `p`). The `release` operation (Figure 5.3) removes an edge from node `p` to node `r` and updates the current priorities using `update`. Adding and removing edges is the responsibility of `acquire` and `release`, the update of the current priorities then is deferred to `update`. `update` traverses the

```
definition acquire :: "fields ref ⇒ fields ref ⇒ unit Heap" where
"acquire p r = do {
    ff.unfold_GrI p;
    ff.unfold_GrI r;
    ff.open_NI r;

    rqs ← fields_lookup fields_qs r;
    ry ← fields_lookup fields_y r;

    (if ry = None
      then do {
        fields_change fields_y' (Some p) r;
        rq ← fields_lookup fields_q r;
        ff.close_NI r rqs;
        ff.fold_GrI r;
        update p (-1) rq }
      else do {
        ff.close_NI r rqs;
        ff.open_NI p;
        fields_change fields_y' (Some r) p;
        pq ← fields_lookup fields_q p;
        pqs ← fields_lookup fields_qs p;
        ff.close_NI p pqs;
        ff.fold_GrI p;
        update r (-1) pq})
  }"
```

Figure 5.2.: Acquire operation of example graph update algorithm

```
definition release :: "fields ref ⇒ fields ref ⇒ unit Heap" where
"release p r = do {
    ff.unfold_GrI p;
    ff.unfold_GrI r;
    ff.open_NI r;
    rq ← fields_lookup fields_q r;
    rqs ← fields_lookup fields_qs r;
    fields_change fields_y' None r;
    ff.close_NI r rqs;
    ff.fold_GrI r;
    update p rq (-1)
  }"
```

Figure 5.3.: Release operation of xample graph update algorithm

```
partial_function (heap) update :: "fields ref ⇒ int ⇒ int ⇒ unit Heap" where
"update n from to = do {
    ff.open_NI n;
    q0 ← fields_lookup fields_q0 n;
    qs ← fields_lookup fields_qs n;
    let qs' = qs - {# from #} + (if to ≥ 0 then {# to #} else {#}) in do {
    fields_change fields_qs' qs' n;
    from ← fields_lookup fields_q n;
    fields_change fields_q' (Max_mset (qs' + {#q0#})) n;
    to ← fields_lookup fields_q n;
    y ← fields_lookup fields_y n;
    ff.close_NI n qs';

    if from ≠ to
      then (case y of
        Some y' ⇒ do {
            ff.unfold_GrI y';
            ff.fold_GrI n;
            update y' from to
          }
        | _ ⇒ do {
          ff.fold_GrI n
        })
      else ff.fold_GrI n
  }
}"
```

Figure 5.4.: Update operation of example graph update algorithm

graph and updates the current priorities along its path.

In the following we will specify and verify the `release` operation using the Flow Framework (and therefore the `update` operation, too). We choose the `release` operation because it allows us not only to demonstrate how to state invariants using the Flow Framework but also how to use this information to additionally prove the termination of `update`. The proof of termination is a new contribution to the PIP example. For the proof of `acquire` please refer to our formalization. That proof is basically analogous to the proof of `release` but has to apply a symmetric invariant (instead of argument `f` being constant for all invocations of `update` argument `t` is constant) and a slightly different termination argument.

5.2.3. Instantiating the Flow Framework

The first step of our verification effort is to instantiate the `FF` locale (see Figure 5.1 for the locale arguments) that we developed in Section 5.1. Note that for time and automation reasons we only use the simple variant of the framework that does not exploit nilpotency or effective acyclicity. This introduces some shortcoming of our program specification that we discuss in Section 5.2.10.

We choose type `int multiset` as our flow domain. Isabelle's standard library already proved that this type is a flow domain, i.e. it is of type-class `cancel_comm_monoid_add`. The mapping from heap objects to nodes is trivial: each PIP node is mapped directly to a node within the Flow Framework, and a set of nodes makes up a graph. The node-local invariant γ for flow graph nodes states that default priorities are non-negative; therefore all values in the predecessor

priority multiset are non-negative, too; the current priority equals the maximum of all predecessor priorities and the local default priority; nodes are not their own successors (i.e. there are no reflexive edges within the graph). Additionally, γ enforces that the multisets of predecessor priorities correspond to the flow in nodes. Finally, γ exposes the successor node using the auxiliary argument `g` in order to state the successor aspect of functional correctness using the auxiliary argument. This implies that $\eta\ x$ represents the successor node of x. Edge functions generated by function `edge` propagate the current priority of nodes to the nodes' (unique) successor nodes. All other edges need to be modelled as 0-functions within the Flow Framework. As γ prohibits that nodes are their own successor it is obvious that `FF`'s assumption `edges_not_refl` holds. The global invariant φ that we use to instantiate the Flow Framework merely enforces that there is neither inflow nor outflow. After instantiation of the Flow Framework, we obtain the predicates `NI` and `GrI` and their accompanying proof rules. The obtained predicates embed PIP nodes into the Flow Framework and enable us to work with and reason about these nodes using the Flow Framework.

5.2.4. Specification of the Operations

We now specify the `release` operation:

```
lemma release_correct:
  assumes "p ∈ X" "r ∈ X" "p ≠ r" " η r = Some p" "i ≠ bot"
    and "∀x ∈ X. case η x of Some y ⇒ y ∈ X | _ ⇒ True"
  shows "<ff.GrI η Fs X i>
    release p r
    <λ_. ∃_AFs'. ff.GrI (η(r := None)) Fs' X i>"
```

If we want to remove an edge $(\mathtt{p},\mathtt{r})$ we require that this edge actually exists in the graph comprising `X`. Furthermore, `p` and `r` must be different and part of the graph `X` and the graph has to have a valid flow interface `i`. The additional assumption that all edges from some node in `X` again point to some node in `X` is stated explicitly to simplify the proof. Alternatively, we could include the global invariant φ `i` as an assumption which in particular implies that there is no outflow, i.e. no edge points outside `X` due to our definition of `edge`. A second alternative would be to extend the node-local invariant γ by a conjunct that ensures that the successor node has to be in `X`. The second alternative is cumbersome to encode into the Flow Framework as we have to inject varying sets `X` into γ using the auxiliary arguments of nodes and graphs and this implies some work to update all nodes and graphs each time `X` changes.

The result of `release` is expected to be a graph still comprising `X` with the same flow interface `i`. However, the auxiliary argument η (representing successor nodes) was updated such that `r` does not have successor `p` any longer. This is the first of the two functional requirements that we expect `release` to satisfy. But the second one is missing at first sight: the current priorities of all nodes in the graph are expected to be adjusted. This requirement is encoded into the node-local invariants in order to avoid a global specification: the change of the current priorities is potentially only bounded by the extent of the graph and we would have to be able and willing to state the footprint of the operation as well as the updated values explicitly. This reasoning also explains why we existentially quantify the resulting heap contents $\mathtt{Fs}'$: we are not able and willing to reason about the extent of the change in $\mathtt{Fs}'$.

Next, we specify `update`:

```
lemma update_correct:
  fixes i ix ixs :: "(fields ref, int multiset) fi"
  assumes "finite (dom_fi i)" "V ⊆ dom_fi i"
```

```
    and "f ∈# inf_fi ix x" "∀x ∈ V. f ∉# fields_qs (Fs x)"
    and "x ∉ xs"
    and "i = ixs + ix''"
    and "ix'' = fi {x}
      (λ_. inf_fi ix x - {# f #} + (if t ≥0 then {# t #} else {#})) (outf_fi ix)"
    and "i ≠ bot" "f > t"
    and "∀x' ∈ {x} ∪ xs. case η x' of Some y ⇒ y ∈ {x} ∪ xs | _ ⇒ True"
  shows "<ff.NI (η x) x (Fs x) m ix * ff.GrI η Fs xs ixs>
    update x f t
    <λ_. ∃A Fs'. ff.GrI η Fs' ({x} ∪ xs) i>"
```

A function invocation `update x f t` is supposed to remove priority `f` from `x`'s set of predecessor priorities and to add priority `t` to `x`'s set of predecessor priorities. If the current priority of `x` was changed by this modification then `update` is supposed to propagate this change to the successor node of `x`.

The first four assumptions of this specifications are related to termination and will be explained later. To simplify the proof we assume $\texttt{x} \notin \texttt{xs}$ which we could also infer from the separating conjunction between node `x` and graph `xs`. Assumption $\texttt{i} = \texttt{ixs} + \texttt{ix}'$ is related to our proof technique, the *fake-interface* approach, which we be explained shortly. $\texttt{i} \neq \bot$ is generally necessary to work with flow interfaces. $\texttt{f} > \texttt{t}$ is necessary to derive that `f` stays constant for all iterations of `update`. Note that we have to prove the correctness of `update` separately depending on whether `update` is invoked by `release` or by `acquire`: while `update` invoked by `acquire` keeps argument `t` constant for all iterations, `update` invoked by `release` keeps argument `f` constant for all iterations. This observation leads to two distinct invariants used to prove termination. The last assumption again ensures that no edge points outside `X`.

5.2.5. PIP in Action

We shortly demonstrate how to apply `acquire` and `release` in Figure 5.5:

```
lemma pip_example_correct:
  shows "<ff.GrI Map.empty Fs0 {} 0 * ↑(φ 0)>
    pip_example
    <λ(a,b,c). (∃A Fs i η. ff.GrI η Fs {a,b,c} i *
                ↑(η = Map.empty(c ↦ b, a ↦ c) ∧
                  i = fi {a,b,c} (λ_. 0) (λ_. 0) ∧ φ i))>"
```

We start with an empty graph that trivially satisfies the global invariant φ. We obtain a graph that contains three nodes $\texttt{a}, \texttt{b}, \texttt{c}$ and two edges $(\texttt{c}, \texttt{a})$ and $(\texttt{a}, \texttt{b})$ and again satisfies the global invariant φ. Additionally, we know that the interface of the graph is $(\{\texttt{a}, \texttt{b}, \texttt{c}\}), 0, 0)$. The resulting graph and its flow interface imply that the current priorities of each of the three nodes are at least the maximum of the default priorities of their predecessors' and its own default priorities (see Section 5.2.10). Note that we do not show that the current priorities are not exactly the maximum of their predecessors' and its own default priorities but only that the current priorities are greater or equal than this maximum. See Section 5.2.10 for a discussion.

Our allocation operation (see Figure 5.6) applies the `replacement` theorem:

```
lemma alloc_correct:
  assumes "q0 ≥ 0" "outf_fi i = (λ_. 0) on -dom_fi i"
  shows "<ff.GrI η Fs X i>
    alloc_node q0
    <λx. ∃A ix. ff.GrI (η(x := None))
      (λx'. if x = x' then Fields None q0 q0 {#} else Fs x') (X ∪ {x}) (i + ix) *
```

```
definition pip_example where
"pip_example ≡ do {
  a ← alloc_node 1;
  b ← alloc_node 2;
  acquire a b;
  c ← alloc_node 3;
  acquire b c;
  release a b;
  acquire c a;

  return (a,b,c)
}"
```

Figure 5.5.: Application of example graph update algorithm

```
definition alloc_node :: "int ⇒ fields ref Heap" where
"alloc_node q0 ≡ do {
  x ← ref (Fields None q0 q0 {#});
  ff.close_NI x ({#} :: int multiset);
  fold_GrI_repl x;
  return x
}"
```

Figure 5.6.: Allocation operation of example graph update algorithm

```
↑(i + ix ≠ bot ∧ ix = fi {x} (λ_. 0) (λ_. 0) ∧ x ∉ dom_fi i)>"
```

For a given default priority `q0` and a graph over nodes X with flow interface i we first allocate a PIP node at address `x` with content `Fields None q0 q0` $\emptyset$, then we close the PIP node and obtain a Flow Framework node with flow interface $i_x = (\{n\}, 0, 0)$. Then we extend graph X using `fold_GrI_repl_rule` to a graph over nodes $\{x\} \cup X$ and interface $i_x + i = (\{x\} \cup X, 0, 0) \neq \bot$. Rule `fold_GrI_repl_rule` is a custom proof rule that in our use case first proves $0 \preccurlyeq i_x$ and then applies the `replacement` theorem to extend flow interface $i = 0 + i$ to flow interface $i_x + i \neq \bot$ (we already demonstrated this approach in Section 4.1.7). This information then is applied to fold node `x` into graph X using `fold_GrI_rule`. In order to apply the `replacement` theorem we had to introduce the assumption on i's outflow in lemma `alloc_correct`.

An approach to prove the existence of flows in programs is to start with trivial flows that correspond to initial states of data structures, e.g. empty data structures. Those should usually be easy to prove. In our example the initial flow graph is 0. For each data structure a valid flow graph is supposed to exist for the entire life time as proving the existence of flows for arbitrary data structures is non-trivial. To modify data structures we decompose the graph of the data structure to obtain the required (minimal) excerpt of the data structure. By iteratively decomposing a graph to an excerpt h_n we obtain a decomposition of the flow graph, e.g. $h = h_1 + (h_2 + (\ldots + (h_{n-1} + h_n))) \neq \bot$. This flow graph decomposition later guides our recomposition. Then, we have at least three options: first, the simple one merely applies a flow interface preserving modification and the excerpt trivially recomposes with the remainder by lemma `int_fg_fi_hom`. This option does not allow us to extend the data structure. The second option applies theorem `replacement` and also allows extending the data structure: we have to prove an initial instance of a contextual extension $\mathsf{int}\, h_n \preccurlyeq \mathsf{int}\, h'_n$ for the modified flow graph h'_n. Iterative application of the replacement theorem finally provides us $h' = h_1 + (h_2 + (\ldots + (h_{n-1} + h'_n))) \neq \bot$. For trivial extensions like

in our PIP example we choose $h_n = 0$ and $h'_n = (\{x\}, \ldots, \ldots)$. Option three works analogously to option two but instead applies theorem `matintain_eff_acyclic_hom` and subflow-preserving extensions.

The next section shows us that there are also far more flexible and creative approaches of working with flow graphs.

5.2.6. Proof Technique: Fake-Interfaces

In `update` we apply a proof technique called fake-interfaces [Krishna et al., 2020, ch. 3]. This technique enables local reasoning for iterative algorithms that achieve their total effect by applying many small changes. Our graph update algorithm is a good example for such an algorithm: it updates the current priority of one node after another, by "fixing" one node its successor node might break and has to be fixed, too. This process is iterated until there is nothing more to be fixed.

The primary difficulty of working with flow graphs is to prove that the flow equation still holds after modifications. Consider `release`: in order to remove edge $(\mathtt{p}, \mathtt{r})$ we have to update the successor field of $\mathtt{p}$. Hence, we unfold $\mathtt{p}$ from $\mathtt{GrI}(\eta, \mathtt{X}, \mathtt{i})$[1] and obtain a graph $\mathtt{GrI}(\eta, \mathtt{X} - \{\mathtt{p}\}, \mathtt{i}')$ and a node $\mathtt{NI}(\eta\ \mathtt{p}, \mathtt{x}, \mathtt{i_p})$ with $\mathtt{i} = \mathtt{i_p} + \mathtt{i}'$. Modifying $\mathtt{p}$ yields a node $\mathtt{NI}(\mathtt{None}, \mathtt{p}, \mathtt{i}'_\mathtt{p})$ where $\mathtt{i}'_\mathtt{p} = (\{\mathtt{p}\}, \mathsf{inf}\ \mathtt{i_p}, 0)$. It does not hold that $\mathtt{i} = \mathtt{i}'_\mathtt{p} + \mathtt{i}'$ as $\mathtt{p}$ no longer provides flow to node $\mathtt{r}$ and the definition of flow interface addition then requests the outside of $\mathtt{X}$ to fill in this expected flow to $\mathtt{r}$ for $\mathtt{p}$, i.e. $\mathtt{i}'_\mathtt{p} + \mathtt{i}'$ is now supposed to additionally contribute the inflow to $\mathtt{i}'$ that was previously provided by $\mathtt{i_p}$. Therefore $\mathtt{i}'_\mathtt{p} + \mathtt{i}' \neq \mathtt{i}$. As we want our total interface to equal $\mathtt{i}$ again (because $\mathtt{i}$ satisfies the global invariant φ and we want to reestablish a valid state of our data structure) we have to adjust $\mathtt{i}'$ to no longer include the flow from $\mathtt{p}$.

If we wanted to incorporate this adjustment at once we encounter the issue of footprints of unknown extent (as we already observed in our specification of `release`): we do not know how to determine or are not willing to determine the footprint and related changes in the flow graph. The key insight of the fake-interface approach is to not try to adjust $\mathtt{i}'$ instantly but to update the interface in lockstep with further computations by deferring flow graph adjustments to later steps. In our example, this approach limits the footprint in each step to two nodes: we unfold $\mathtt{r}$ from $\mathtt{GrI}(\eta, \mathtt{X} - \{\mathtt{p}\}, \mathtt{i}')$ and obtain $\mathtt{NI}(\eta\ \mathtt{r}, \mathtt{r}, \mathtt{i_r})$ and $\mathtt{GrI}(\eta, \mathtt{X} - \{\mathtt{p}, \mathtt{r}\}, \mathtt{i}'')$ with $\mathtt{i}' = \mathtt{i_r} + \mathtt{i}''$, i.e. $\mathtt{i} = \mathtt{i_p} + \mathtt{i_r} + \mathtt{i}''$. If we find some $\mathtt{i}'_\mathtt{r}$ such that $\mathtt{i_p} + \mathtt{i_r} = \mathtt{i}'_\mathtt{p} + \mathtt{i}'_\mathtt{r}$ then we obtain $\mathtt{i} = \mathtt{i}'_\mathtt{p} + \mathtt{i}'_\mathtt{r} + \mathtt{i}''$. This way we can fold $\mathtt{p}$ back into $\mathtt{GrI}(\eta, \mathtt{X} - \{\mathtt{p}, \mathtt{r}\}, \mathtt{i}'')$ and obtain $\mathtt{GrI}(\eta, \mathtt{X} - \{\mathtt{r}\}, \mathtt{i}'_\mathtt{p} + \mathtt{i}'')$. By doing this we break composability of this graph with $\mathtt{r}$ as $\mathtt{r}$ still has flow interface $\mathtt{i_r}$ while $\mathtt{i_p} + \mathtt{i}'$ is only known to compose with $\mathtt{i}'_\mathtt{r}$. Restoring composability of $\mathtt{r}$ by making its interface equal $\mathtt{i}'_\mathtt{r}$ then is delegated to `update`.

The steps executed by `update` in principle follow the same scheme as the first step executed by `release`: update the current node to make its interface satisfy the expected shape, rewrite the interfaces of the current node and its successor node in order to become able to fold back the current node into the graph, then repeat this process for the now "broken" successor node. This process terminates when there is no more successor or change of the flow graph.

In our example the flow interface rewriting $\mathtt{i_p} + \mathtt{i_r} = \mathtt{i}'_\mathtt{p} + \mathtt{i}'_\mathtt{r}$ in `release` is done as follows: we know that $\mathtt{i_p} = (\{\mathtt{p}\}, \lambda\_.\ \mathtt{m_p}, \lambda \mathtt{y}.\ \mathtt{edge\ p\ fs_p\ y\ m_p})$ for some $\mathtt{m_p}$ and $\mathtt{fs_p}$, and $\mathtt{i}'_\mathtt{p} = (\{\mathtt{p}\}, \lambda\_.\ \mathtt{m_p}, 0)$, i.e. the flow from $\mathtt{p}$ to $\mathtt{r}$ vanishes and we have to remove $\mathtt{p}$'s contribution $C := \mathtt{edge\ p\ fs_p\ r\ m_p}$ from $\mathtt{r}$'s flow $\mathtt{m_r}$. We obtain

$$\mathtt{i_p} + \mathtt{i_r} = (\{\mathtt{p}\}, \lambda\_.\ \mathtt{m_p}, 0) + (\{\mathtt{r}\}, \lambda\_.\ \mathtt{m_r} \setminus \{\mathtt{C}\}, \lambda \mathtt{y}.\ \mathtt{edge\ r\ fs_r\ y}\ (\mathtt{m_r} \setminus \{\mathtt{C}\}))$$

[1] We omit the `Fs` and `fs` arguments to simplify our presentation.

$$\frac{\bigwedge \texttt{a F.}\ \dfrac{\texttt{a} \in \texttt{F}}{\texttt{P a F}} \qquad \bigwedge \texttt{x F'.}\ \dfrac{\texttt{finite F} \qquad \texttt{F'} \subseteq \texttt{F} \qquad \texttt{F'} \subseteq \texttt{F} \qquad \texttt{x} \in \texttt{F} \qquad \bigwedge \texttt{y.}\ \dfrac{\texttt{y} \in \texttt{F}}{\texttt{P y (\{x\} } \cup \texttt{ F')}}}{\texttt{P x F'}} \qquad \texttt{a} \in \texttt{F}}{\texttt{P a F'}}$$

Figure 5.7.: Induction rule that simplifies the termination proof of the update operation

The crucial observation to make the fake-interface approach work is that we can keep around parts of graphs that are supposed to be composed but have to be adjusted in order to do so.

5.2.7. Termination

A technical difficulty of the proof of update within Isabelle is that Imperative-HOL's separation logic framework [Lammich and Meis, 2012] does not support partial correctness. Therefore, we also prove termination for `update` as an additional contribution. The key observation for the termination proof of `update` is that as soon as an already visited node is reencountered then the algorithm terminates, i.e. if update reencounters a node during traversal we can prove that the guard around the recursive call can not be satisfied and therefore we do not have to consider a recursive call.

To separate the termination issue from the original verification issue we introduce an induction rule (see Figure 5.7) that is inspired by the corresponding graph traversal narrative: we show inductively that the statement to prove holds for *shrinking* sets of already visited nodes. This is motivated by the above observation that only a single node is reencountered at most once: if we already visited all nodes F then the next encountered node x was already visited, i.e. $x \in F$, and we can prove that the algorithm terminates, i.e. we do not need an induction hypothesis and therefore this case is the base case. The step case assumes that there is some set $F' \subseteq F$ of already visited nodes and we are currently visiting a node x. We obtain the induction hypothesis that the statement under proof already holds for a larger set of already visited nodes $\{x\} \cup F'$ and an arbitrary node y being visited next. Note that in the step case we do not specify if the currently visited node $x \in F$ was already visited or not, i.e. if $x \in F'$ holds or not. Therefore, we have to consider both possibilities in our proofs using this induction rule.

We apply this induction rule in the following proof of `update` using $F \coloneqq \mathsf{dom}\, h$ (the upper bound of visitable nodes is exactly the set of nodes within the flow graph). To prove that we do not call `update` recursively in the base case we have to accumulate some information along the path of already visited nodes. In the case of a call `update x` (-1) `t` by `acquire` we have to accumulate for all already visited nodes $x \in F'$ that `t` $\in$ `fields_qs` x, and for a call `update x f` (-1) by `release` we have to accumulate for all already visited nodes $x \in F'$ that `f` $\notin$ `fields_qs` x. These invariants build upon the insight that during an `update` traversal triggered by `acquire` argument `t` stays constant for all recursive calls and analogously for an `update` traversal triggered by `release` argument `f` stays constant. In the `acquire` case this information is already sufficient to prove together with `t` $>$ `f` that for the second encounter of a node there is no change in the current priority and therefore no recursive call is executed. In the `release` case we additionally need to propagate the information along the path that `f` $\in$ `fields_qs` x for the currently visited node x. This fact is implied for each step by the guard around the recursive call in `update`, `t` $<$ `f` and the flow equation. In the base case these facts provide us immediately a contradiction, i.e. it is actually impossible for `update` invoked by

`release` to reencounter an already visited node.[2]

5.2.8. Example Proof

We prove the correctness of `update` for the `release`-case:

```
lemma update_correct:
  fixes i ix ixs :: "(fields ref, int multiset) fi"
  assumes "finite (dom_fi i)" "V ⊆ dom_fi i"
    and "f ∈# inf_fi ix x" "∀x ∈ V. f ∉# fields_qs (Fs x)"
    and "x ∉ xs"
    and "i = ixs + ix''"
    and "ix'' = fi {x}
      (λ_. inf_fi ix x - {# f #} + (if t ≥0 then {# t #} else {#})) (outf_fi ix)"
    and "i ≠ bot" "f > t"
    and "∀x' ∈ {x} ∪ xs. case η x' of Some y ⇒ y ∈ {x} ∪ xs | _ ⇒ True"
  shows "<ff.NI (η x) x (Fs x) m ix * ff.GrI η Fs xs ixs>
    update x f t
    <λ_. ∃A Fs'. ff.GrI η Fs' ({x} ∪ xs) i>"
```

Proof. We summarize our formal proof using a Hoare-logic style proof. To avoid significant duplication of facts we do not repeat pure facts (printed in blue). Those are valid for the entire remainder of the proof once introduced. Spatial facts are always repeated and printed in red.

We conduct our proof by induction over the set of already visited nodes `V` using induction rule `finite_induct_reverse` for arbitrary $\mathtt{t}, \mathtt{X}, \mathtt{i_x}, \mathtt{i_X}, \mathtt{Fs}, \mathtt{m}, \eta$.

First we consider the base case: we know that the current node x was already visited, i.e. $\mathtt{x} \in \mathtt{V}$. Therefore, we know by assumption that both $\mathtt{f} \in \mathtt{inf\_fi\ i_x\ x}$ and $\mathtt{f} \notin \mathtt{fields\_qs\ (Fs\ x)}$. Additionally, we obtain $\gamma\ (\eta\ \mathtt{n})\ \mathtt{x}\ (\mathtt{Fs\ x})\ \mathtt{m}$ and $\mathtt{inf\_fi\ i_x\ x} = \mathtt{m}$ from $\mathtt{NI}\ (\eta\ \mathtt{x})\ \mathtt{x}\ \mathtt{m}\ \mathtt{i_x}$. Furthermore, $\gamma\ (\eta\ \mathtt{x})\ \mathtt{x}\ (\mathtt{Fs\ x})\ \mathtt{m}$ implies that $\mathtt{m} = \mathtt{fields\_qs\ (Fs\ x)}$. Therefore, we obtain the contradiction $\mathtt{f} \in \mathtt{m}$ and $\mathtt{f} \notin \mathtt{m}$ and finished the base case of this proof.

Next we consider the step case:

```
1  update x from to = do {
2      { x ∉ X * to < from * i = i''x + iX ≠ ⊥ * (∀x' ∈ {x} ∪ X. case η x' of Some y ⇒ y ∈ {x} ∪ X) *
         Fs0 x = (yx, qx, q0x, qsx) *
         i''x = ({x}, λ_. qsx − {from} + (if to ≥ 0 then {to} else ∅), λy. edges x (Fs0 x) y qsx) *
         V ⊆ dom_fi i * from ∈ inf_fi ix x * (∀x' ∈ V. from ∉ fields_qs (Fs x')) *
         NI (η x) x (Fs0 x) qsx ix * GrI η Fs0 X iX }
3      ff.open_NI x;
4      { γ (η x) x (Fs0 x) qsx * ix = ({x}, λ_. qsx, λy. edges x (Fs0 x) y qsx) *
         x ↦ Fs0 x * GrI η Fs0 xs iX }
5      q0 <- fields_lookup fields_q0 x;
6      qs <- fields_lookup fields_qs x;
7      let qs' = qs - {# from #} + (if to >= 0 then {# to #} else {#}) in do {
8      fields_change fields_qs' qs' x;
```

[2] If we removed an edge that is part of a cycle then we broke the cycle before calling `update` and therefore will not be able to traverse a cycle. Otherwise we removed an edge that is not part of a cycle but could lead to a cycle. We distinguish two cases: first, the priority that is to be removed by update equals the current priority of all nodes in the cycle. Then we already terminate when we encounter the cycle as the dominant priority is contained at least twice in the set of predecessor priorities and we only remove a single one of them, i.e. the current priority does not change. Second, the priority that is to be removed by update is less than the current priority in the cycle. Then we terminate also when we encounter the first node of the cycle as the current priority does not change.

```
9   from' <- fields_lookup fields_q x;
10  fields_change fields_q' (Max_mset (qs' + {#q0#})) x;
11  to' <- fields_lookup fields_q x;
12  y <- fields_lookup fields_y x;
13  { q0 = q0_x * qs = qs_x * qs' = qs_x − {from} + (if to ≥ 0 then {to} else {}) *
      from' = q_x * q' = max (qs' + {q0}) * to' = q' * y = y_x *
      Fs_1 = λx'. if x' = x then (y_x, q', q0_x, qs') else Fs_0 x' *
      x ↦ Fs_1 x * GrI η Fs_1 xs i_X }
14  ff.close_NI x qs';
15  { i'_x = ({x}, λx. qs', λx'. edge x (Fs_1 x) x' qs') *
      NI (η x) x (Fs_1 x) qs' i'_x * GrI η Fs_1 xs i_X }
16
17  if from' != to'
18    then (case y of
19      Some y' => do {
20          { y = Some y' * from' ≠ to' *
              NI (η x) x (Fs_1 x) qs' i'_x * GrI η Fs_1 xs i_X }
21          ff.unfold_GrI y';
22          { i_X = i_y' + i'_X * i_y' = ({y'}, λx. qs_y', λx. edge y' (Fs_1 y') x qs_y') *
              Fs_1 y' = (y_y', q_y', q0_y', qs_y') *
23            NI (η x) x (Fs_1 x) qs' i'_x * NI (η y') y' (Fs_1 y') qs_y' i_y' * GrI η Fs_1 (X − {y'}) i'_X }
24          { i = i''_x + i_y' + i'_X = i'_x + i'_y' + i'_X *
              i'_y' = ({y'}, λ_. qs_y' − {from'} + {to'}, λx. edge y' (Fs_1 y') x qs_y') *
25            NI (η x) x (Fs_1 x) qs' i'_x * NI (η y') y' (Fs_1 y') qs_y' i_y' * GrI η Fs_1 (X − {y'}) i'_X }
26          ff.fold_GrI x;
27          { NI (η y') y' (Fs_1 y') qs_y' i_y' * GrI η Fs_1 (X − {y'} + {x}) (i'_x + i'_X) }
28          { y' ∉ X − {y'} + {x} * to' < from' * i = i'_y' + (i'_x + i'_X) ≠ ⊥ *
              (∀x ∈ {y'} ∪ (X − {y'} + {x}). case η x of Some y ⇒ y ∈ {x} ∪ X) *
              i'_y' = ({y'}, λ_. qs_y' − {from'} + (if to' ≥ 0 then {to'} else ∅), λy. edges y' (Fs_1 y') y qs_y') *
              V ∪ {x} ⊆ dom_fi i * from' ∈ inf_fi i_y' y' * (∀x ∈ V ∪ {x}. f ∉ fields_qs (Fs x)) *
29            NI (η y') y' (Fs_1 y') qs_y' i_y' * GrI η Fs_1 (X − {y'} + {x}) (i'_x + i'_X) }
30          update y' from to'
31          { GrI η Fs_2 ({x} ∪ X) i }
32        }
33      | _ => do {
34          { y = None * from' ≠ to' *
              NI (η n) x (Fs_1 x) qs' i'_x * GrI η Fs_1 xs i_X }
35          { i'_x = i''_x *
              NI (η x) x (Fs_1 x) qs' i'_x * GrI η Fs_1 xs i_X }
36          ff.fold_GrI x
37          { GrI η Fs_1 ({x} ∪ X)) i }
38        })
39    else do {
40      { from' = to' *
```

```
            NI (η x) x (Fs₁ x) qs' i'ₓ ∗ GrI η Fs₁ xs iX }
41        { i'ₓ = i''ₓ ∗
            NI (η x) x (Fs₁ x) qs' i'ₓ ∗ GrI η Fs₁ xs iX }
42        ff.fold_GrI x
43        { GrI η Fs₁ ({x} ∪ X)) i }
44      }
45    }
46  }
```

We start we a summary of all assumptions in Line 2. In Line 3 we open node `x` to obtain access to the underlying heap object in order to modify the PIP node. Line 13 summarizes all changes applied in lines 5 to 12. In Line 14 we close node `x` again and obtain the induced flow interface $\mathtt{i}'_\mathtt{x}$. Now we have to consider three cases: $\mathtt{from}' \neq \mathtt{to}' \wedge \mathtt{y_x} \neq \mathtt{None}$, and $\mathtt{from}' \neq \mathtt{to}' \wedge \mathtt{y_x} = \mathtt{None}$, and $\mathtt{from}' = \mathtt{to}'$.

We start with case $\mathtt{from}' \neq \mathtt{to}' \wedge \mathtt{y_x} \neq \mathtt{None}$ in Line 20. Analogously to the base case we can infer that we have not visited node x yet, i.e. $\mathtt{x} \notin \mathtt{V}$. We have some $\mathtt{y}'$ such that $\mathtt{y_x} = \mathtt{Some}\ \mathtt{y}'$. We unfold node $\mathtt{y}'$ in Line 21. Here we know that $\eta\ x = \mathtt{y}' \in X$ due to our assumption that all edges stay within the graph. Then we manipulate the flow interfaces of `x` and $\mathtt{y}'$: we know that $\mathtt{i''_x} + \mathtt{i_{y'}} \neq \bot$. $\mathtt{i''_x}$ has an outflow of $\{\mathtt{from}'\}$ to $\mathtt{y}'$, while $\mathtt{i}'_\mathtt{x}$ has an outflow of $\{\mathtt{to}'\}$. Therefore, we have to remove $\mathtt{from}'$ from $\mathtt{y}'$'s inflow and add $\mathtt{to}'$ to its inflow to compensate for the change in `x`'s outflow and to obtain flow interface $\mathtt{i}'_{\mathtt{y}'}$ such that $i''_x + i_{y'} = i'_x + i'_{y'}$ (see lemma `adjust_interface_mset'` in our formalization). We substitute this equality into the $\mathtt{i} = \mathtt{i''_x} + \mathtt{i_{y'}} + \mathtt{i'_X}$ and obtain $\mathtt{i} = \mathtt{i}'_\mathtt{x} + \mathtt{i}'_{\mathtt{y}'} + \mathtt{i}'_\mathtt{X} \neq \bot$. This implies $i'_x + i'_X \neq \bot$ and we can fold node `x` into graph $X - \{y'\}$. In Line 28 we establish the precondition for the recursive call. Non-trivial facts thereof are:

- $\mathtt{to}' < \mathtt{from}'$: we know that both $\mathtt{to}' = \mathtt{max}\ (\{\mathtt{q0}\} \cup (\mathtt{m} - \{\mathtt{from}\} + \{\mathtt{to}\}))$ and $\mathtt{from}' = \mathtt{max}\ (\{\mathtt{q0}\} \cup \mathtt{m})$. Furthermore, $\mathtt{to} < \mathtt{from}$ and $\mathtt{from} \in \mathtt{m}$. Therefore, $\mathtt{to}' \leq \mathtt{from}'$. Our case distinction already provided us with $\mathtt{to}' \neq \mathtt{from}'$. Therefore, $\mathtt{to}' < \mathtt{from}'$.

- $\mathtt{from} \in \mathtt{inf\_fi}\ \mathtt{i_{y'}}\ \mathtt{y}'$: We have $i''_x + i_{y'} \neq \bot$. The definition of flow interface sums provides us that $\mathtt{inf\_fi}\ \mathtt{i_{y'}}\ \mathtt{y}' = \mathtt{inf\_fi}\ (\mathtt{i''_x} + \mathtt{i_{y'}})\ \mathtt{y}' + \mathtt{outf\_fi}\ \mathtt{i''_x}\ \mathtt{y}'$. We know that for arbitrary flows `m` it holds that $\mathtt{outf\_fi}\ \mathtt{i''_x}\ \mathtt{y}'\ \mathtt{m} = \mathtt{edge}\ \mathtt{x}\ (\mathtt{Fs_1}\ \mathtt{x})\ \mathtt{y}'\ \mathtt{m} = \mathtt{max}\ (\{\mathtt{q0_x}\} \cup \mathtt{m})$. Moreover, $\mathtt{from}' \neq \mathtt{to}'$ and $\mathtt{to} < \mathtt{from}$ imply that $\mathtt{max}\ \mathtt{m} = \mathtt{from}$. Moreover, $\mathtt{q0_n} < \mathtt{from}$ as otherwise $\mathtt{from}' = \mathtt{to}'$. Combining the previous two results yields $\mathtt{from} = \mathtt{max}\ (\{\mathtt{q0_m}\} \cup \mathtt{m}) = \mathtt{from}'$. This provides us together with the above result that $\mathtt{from} \in \mathtt{outf\_fi}\ \mathtt{i''_x}\ \mathtt{y}' \subseteq \mathtt{inf\_fi}\ \mathtt{i_{y'}}\ \mathtt{y}'$.

- $\mathtt{f} \notin \mathtt{fields\_qs}\ (\mathtt{Fs_1}\ \mathtt{x})$: if there was $\mathtt{f} \in \mathtt{fields\_qs}\ (\mathtt{Fs_1}\ \mathtt{x})$ then we would have obtained $\mathtt{from}' = \mathtt{to}'$ which contradicts our assumption $\mathtt{from}' \neq \mathtt{to}'$.

The next line executes the recursive call. In order to do so we have to show $\mathtt{from}' = \mathtt{from}$: we showed this already in the second item of our derivation of non-trivial assumptions. The recursive call provides us just the result we have to show ourselves.

In Line 35 we show $\mathtt{i}'_\mathtt{x} = \mathtt{i''_x}$ for case $\mathtt{from}' \neq \mathtt{to}' \wedge \mathtt{y} = \mathtt{None}$. First we show that the outflows of $\mathtt{i}'_\mathtt{x}$ and $\mathtt{i''_x}$ agree. We know that $\mathtt{y} = \mathtt{y_x} = \mathtt{fields\_y}\ (\mathtt{Fs_0}\ \mathtt{x}) = \mathtt{fields\_y}\ (\mathtt{Fs_1}\ \mathtt{x}) = \mathtt{None}$. Then, for all $\mathtt{x}'$ holds

$$\begin{aligned}
\mathtt{edges}\ \mathtt{x}\ (\mathtt{Fs_1}\ \mathtt{x})\ \mathtt{x}'\ \mathtt{qs}' &= \mathtt{if\ Some}\ \mathtt{x}' = \mathtt{None\ then}\ _\ \mathtt{else}\ \emptyset \quad \text{by } \mathtt{fields_y}\ (\mathtt{Fs_1}\ \mathtt{x}) = \mathtt{None} \\
&= \emptyset \\
&= \mathtt{if\ Some}\ \mathtt{x}' = \mathtt{None\ then}\ _\ \mathtt{else}\ \emptyset
\end{aligned}$$

$$= \texttt{edges x } (\texttt{Fs}_0\ \texttt{x})\ \texttt{x qs}_\texttt{x} \qquad \text{by } \texttt{fields\_y } (\texttt{Fs}_0\ \texttt{x}) = \texttt{None}$$

Domain and inflow are trivially identical. Therefore, $\texttt{i}'_\texttt{x} = \texttt{i}''_\texttt{x}$. Using this we can fold node `x` into graph `X` in Line 36.

Similarly, in Line 41 we show $\texttt{i}'_\texttt{x} = \texttt{i}''_\texttt{x}$ for case $\texttt{from}' = \texttt{to}'$ by an additional case distinction: If $\texttt{y}_\texttt{x} \neq \texttt{Some y}'$ then we are done analogously to the previous case. If $\texttt{y}_\texttt{x} = \texttt{Some y}'$ then we observe that

$$\texttt{max } (\texttt{qs}_\texttt{x} + \{\texttt{q0}_\texttt{x}\}) = \texttt{max } ((\texttt{qs}_\texttt{x} - \{\texttt{from}\} + (\texttt{if to} \geq 0\ \texttt{then } \{\texttt{to}\}\ \texttt{else } \emptyset)) + \{\texttt{q0}_\texttt{x}\})$$

Using this and $\texttt{y}_\texttt{x} = \texttt{Some y}'$ we obtain

$$\begin{aligned}
\texttt{edges x } (\texttt{Fs}_1\ \texttt{x})\ \texttt{y}'\ \texttt{qs}' &= \texttt{if Some y}' = \texttt{y}_\texttt{x}\ \texttt{then max } (\texttt{qs}' \cup \{\texttt{q0}_\texttt{x}\})\ \texttt{else } \emptyset \\
&= \texttt{max } ((\texttt{qs}_\texttt{x} - \{\texttt{from}'\} + (\texttt{if to} \geq 0\ \texttt{then } \{\texttt{to}\}\ \texttt{else } \emptyset)) \cup \{\texttt{q0}_\texttt{x}\}) \\
&= \texttt{max } (\texttt{qs}_\texttt{x} \cup \{\texttt{q0}_\texttt{x}\}) \\
&= \texttt{if Some y}' = \texttt{y}_\texttt{x}\ \texttt{then max } (\texttt{qs}_\texttt{x} \cup \{\texttt{q0}_\texttt{x}\})\ \texttt{else } \emptyset \\
&= \texttt{edges x } (\texttt{Fs}_0\ \texttt{x})\ \texttt{y}'\ \texttt{qs}_\texttt{x}
\end{aligned}$$

■

5.2.9. Implementation

The verification of the PIP example using our implementation of the Flow Framework basically follows the proof outline in the previous subsection. The proof rules introduced by our implementation of the Flow Framework enable efficient and effective injection of proof steps into automatic proof methods of our Separation Logic framework. However, there are several difficulties remaining in our proofs (not only related to the Flow Framework) that should be addressed by further development:

- The verifying condition generator introduces a case distinction $\texttt{to} \geq 0$ in operation `update`. This case distinction duplicates the efforts required in our proof. Slight differences in both proofs imposed a significant amount of additional effort.
- For some unknown reason we were not able to state a lemma about the flow shift required by fake interfaces that enables the simplifier to automatically solve flow interface equations that contain this shift. The implicit nature of solutions to sums of flow interfaces probably requires significant effort to design effective proof methods or sets of simplification rules. Multiple ideas for simplification rules during the course of this thesis only yielded insignificant improvement.
- Similarly, we have to repeatedly show that flow graph sums are valid by using the validity of larger sums. These mostly required manual intervention, especially if additionally the fake interface flow shift is involved. Sometimes, non-trivial reasoning on multisets additionally complicated these steps.
- Proof steps that can not be solved by proof method `sep_auto` introduce significant delays in the runtime of the proofs as `sep_auto` tries for too long to solve these steps before falling back to human assistance. Therefore, the proofs of the PIP example are very slow.
- Our opaque node and graph abstractions introduced by the Flow Framework prevent spatial reasoning usually enabled by Separation Logic, e.g. that two pointers are different. We had to introduce dedicated lemmas that enables this reasoning for our abstractions. These rules are not yet integrated into the automated proof methods and therefore require manual steps.

5.2.10. Thoughts About Invariants

The previous example expresses some invariant in graphs using the Flow Framework. The original PIP invariant was stated in a global fashion as "the priority of each node corresponds to the maximum of its default priority and its predecessors' default priorities". How can we be sure that our local invariant is equivalent to the global one?

There are two points of view on this question: first, if the invariant is merely a means of proving some greater property then we can take the stance that it does not matter if both invariants are equal as long as the chosen invariant enables us to prove the property. In this case we could also call the invariant a technical detail of the proof. In our example the involvement of the flow to prove termination of update in the release case can be seen as an instance for this point of view: we do not care how termination was proven, the only thing that matters is that is was proven.

However, regarding the invariant about priorities in our algorithm we can not apply this reasoning. Our graph update algorithm is only a component of the PIP. To plug our algorithm into the PIP we have to either show that our algorithm satisfies the specification that the PIP expects from priority bookkeeping algorithms or otherwise we have to prove the entire PIP using the specification we introduced in our algorithm. The second option is again an instance of our first point of view while the first option requires us to prove the equivalence of the local and the global invariant.

In our example we follow the second option and show that in each node y the current priority `fields_y (Fs y)` is greater or equal the maximum of its own and all predecessors' default priorities (along finite paths):

```
lemma upper_bound:
  assumes "finite P" "X = dom_fg H" "y ∈ X" "P ⊆ paths Fs X y" "H ≠ bot"
  shows "ff.Gr η Fs X H ⟹_A ff.Gr η Fs X H *
    ↑(fields_q (Fs y) ≥
      Max ({fields_q0 (Fs (hd xs)) | xs. xs ∈ P } ∪ {fields_q0 (Fs y)}))"
```

Note that we can assume the validity of flow graph H as maintaining a valid flow graph for the entire live time of a data structure is essential in the use of the flow framework as proving flows for arbitrary data structures is non-trivial (remember our discussion in Section 5.2.5). Furthermore, note that we do not show that this upper bound is tight. In fact, it is not: the local invariant also holds for cycles with a self-sustaining priority that is not founded by a default priority of a participating node, i.e. the flow for a graph and inflow is not unique in this case. To prevent such self-sustaining flow contributions we would have to apply effective acyclicity (which unfortunately exceeded the current level of automation of our Flow Framework implementation and remaining time for this thesis).

Now, the original purpose of the Flow Framework was to avoid global reasoning and yet we are again unable to avoid it in the case of the second point of view. However, while reasoning globally we avoided the "splitting issue" described in Chapter 2 by breaking our verification down into two steps: first, prove the local invariant using the approach that avoids the "splitting issue", and second, relate the local invariant to the global invariant using information only that does not suffer from the "splitting issue", i.e. in our proof of `upper_bound` we have unrestricted access to the entire node-local invariant without caring about spatial constraints imposed by Separation Logic.

6. Proving Theorem 3

In this section we try to give a consistent paper-version of our formalization of [Krishna et al., 2020, theorem 3]. There is an existing high-level proof in [Krishna, 2019, th. 3.38] that we refine in this chapter. The goal of this chapter is to provide all details that are relevant to re-implement and enhance our proof without too much struggle.

6.1. Preliminaries

Notation

We introduce an arrow notation

$$m \xrightarrow{xs}_e y \equiv \mathsf{chain}\ \ e\ xs\ y\ m$$

to denote path functions (composition of edge functions along a path) along paths xs to y using edge functions e with an initial flow value m. Flow along concatenated chains xs_i using edge functions e_i is written as

$$m \xrightarrow{xs_1}_{e_1} \xrightarrow{xs_2}_{e_2} \cdots \xrightarrow{xs_{n-1}}_{e_{n-1}} \xrightarrow{xs_n}_{e_n} y$$

This chain corresponds to

$$\begin{aligned}
&\mathsf{chain}\, e_n\ xs_n\ y\ (\\
&\mathsf{chain}\, e_{n-1}\ xs_{n-1}\ (\mathsf{hd}\,(\mathsf{concat}\,(xs_n \cdot [y])))\ (\\
&\cdots\\
&\mathsf{chain}\, e_2\ xs_2\ (\mathsf{hd}\,(\mathsf{concat}\,(xs_3 \cdots xs_n \cdot [y])))\ (\\
&\mathsf{chain}\, e_1\ xs_1\ (\mathsf{hd}\,(\mathsf{concat}\,(xs_2 \cdots xs_n \cdot [y])))\ m) \cdots))
\end{aligned}$$

Therefore, if there are empty path segments xs_i then the corresponding path segment function equals id and the destination nodes for path segment functions preceding the empty path segment are obtained from path segments succeeding the empty path segment. This translation enables us to state an equivalence for arrows corresponding to lemma `chain_append_nonempty`, i.e. this equivalence does not impose constraints regarding path segments being nonempty.

$$\xrightarrow{xs \cdot xs'}_e \quad \equiv \quad \xrightarrow{xs}_e \xrightarrow{xs'}_e$$

We use this notation for the proof of theorem 3 in order to simplify the proof. Due to the proof's high abstraction level low-level details like lists being empty or not produce too much noise that obstructs the proof's key ideas.

If there is some designated node x within a path xs, i.e. $xs = xs_1 \cdot x \cdot xs_2$, then we obtain the equivalence

$$m \xrightarrow{xs_1 \cdot x \cdot xs_2}_e \quad \equiv \quad (m \xrightarrow{xs_1}_e x) \xrightarrow{x \cdot xs_2}_e$$

Additionally, we lift the arrow notation to convoluted paths $xss = [xs_1, \ldots, xs_n]$:

$$\xrightarrow{xss}_e \equiv \xrightarrow{xs_1}_e \cdots \xrightarrow{xs_n}_e$$

We denote capacities in h that approximate paths using the notation

$$m \xrightarrow{xs}\!\!\!\!\!\rightarrow_e y \equiv \mathsf{cap}\, e\ (\mathsf{hd}\, xs)\ y\ m$$

This notation that only uses the heads of paths as source nodes for capacities is motivated by the fact that we will approximate paths and path segments with capacities (if $|xs| < |h|$ then $\mathsf{cap}\, e\ (\mathsf{hd}\, xs)\ y$ includes the flow along xs to y, i.e. $m \xrightarrow{xs}_e y \leq m \xrightarrow{xs}\!\!\!\!\!\rightarrow_e y$). We again lift this notation to chains of capacities:

$$m \xrightarrow{xss}\!\!\!\!\!\rightarrow_e y \equiv m \xrightarrow{xs_1}\!\!\!\!\!\rightarrow_e \cdots \xrightarrow{xs_n}\!\!\!\!\!\rightarrow_e y \equiv \mathsf{chains}'\ (\lambda xs\ y.\ \mathsf{cap}\, e\ (\mathsf{hd}\, xs)\ y)\ xss\ y\ m$$

If the annotated function f is not an edge function then we consider

$$m \xrightarrow{xss}\!\!\!\!\!\rightarrow_f y \equiv m \triangleright f\ xs_1\ (\mathsf{hd}\, xs_2) \triangleright \cdots \triangleright f\ xs_n\ y \equiv \mathsf{chains}'\ f\ xss\ y\ m$$

Another notational convention is that if the value in front of an arrow is a function from nodes to flow values then we obtain the initial flow value by applying the first node along the path to this function:

$$f \xrightarrow{xs}_e y \equiv f\ (\mathsf{hd}\, (xs \cdot y)) \xrightarrow{xs}_e y$$

This notation avoids some pitfalls and notational inconvenience if the first path segments are empty. For concatenated chains this translates to:

$$f \xrightarrow{xs_1}_{e_1} \cdots \xrightarrow{xs_n}_{e_n} y \equiv f\ (\mathsf{hd}\, (xs_1 \cdots x_n \cdot y)) \xrightarrow{xs_1}_{e_1} \cdots \xrightarrow{xs_n}_{e_n} y$$

Implicit Assumptions

In this chapter the following assumptions apply implicitly:

- The flow domain is a positive cancellative commutative monoid.
- There is a reduced closed set of endomorphisms E.
- $(\lambda\_.\ 0) \in E$, $\mathrm{id} \in E$
- All edge functions of all flow graphs are drawn from E.

An immediate conclusion from these assumptions is that all arrow notations from the above notation are defined (the lemmas underlying the arrow equalities usually require endomorphic edge functions) and that E is closed under arrows.

For each flow graph h we assume the existence of its accompanying domain N, edge functions e, and flow f. E.g. for a flow graph h_7'' there are also its components N_7'', e_7'' and f_7''. For each flow graph h we denote its flow interface with i.

6.2. Abstractions

In the proof of theorem 3 we will often apply similar proof steps on multiple levels of abstraction. The pattern usually is related to following some inflow along a path within a graph, and also following some inflow along a path through multiple subgraphs of a sum graph.

A general reasoning pattern in theorem 3 is to show that some sum over path functions does not equal zero and therefore, there must be a path that does not equal zero. To streamline this reasoning style we construct a set of abstractions that over-approximate given terms and enable us to chain multiple steps, e.g.:

$$0 < \ldots \leq \ldots \leq \ldots = \ldots \leq \sum \ldots$$

We provide a small overview of the introduced approximation lemmas to develop some intuition about our approach. For the remaining lemmas and their proofs, please refer to our formalization.

We start with simple statements connecting inflow, outflow and flow in a single node x:

```
lemma inf_fg_le_inf_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h1 + h2 ≠ bot" "x ∈ dom_fg h1" "∀x y. edge_fg h1 x y ∈ E"
  shows "inf_fg (h1 + h2) x ≤ inf_fg h1 x"

lemma inf_fg_le_flow_fg:
  fixes h :: "('n,'m :: pos_cancel_comm_monoid_add) fg"
  assumes "h ≠ bot" "n ∈ dom_fg h"
  shows "inf_fg h n ≤ flow_fg h n"

lemma outf_fg_le_inf_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "h = h1 + h2" "x ∈ dom_fg h2" "∀x y. edge_fg h1 x y ∈ E"
  shows "outf_fg h1 x ≤ inf_fg h2 x"
```

These statements immediately follow from the definitions of flow interface sums and the flow equation. `inf_fg_le_inf_fg` and `outf_fg_le_inf_fg` already provide us some insight into the interaction of composed flow graphs.

Next we observe that we can approximate a single edge leaving a flow graph by the entire outflow:

```
lemma edge_fg_flow_fg_le_outf_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "x ∈ dom_fg h" "y ∈ -dom_fg h" "∀x y. edge_fg h x y ∈ E"
  shows "edge_fg h x y (flow_fg h x) ≤ outf_fg h y"
```

This follows immediately from the definition of outflow.

The flow along a path in a flow graph to a destination node y can be approximated by y's flow:

```
lemma chain_inf_fg_le_flow_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "End_closed E" "∀x y. edge_fg h x y ∈ E"
    "set xs ⊆ dom_fg h" "y ∈ dom_fg h" "id ∈ E" "(λ_. 0) ∈ E"
  shows "chain (edge_fg h) xs y (inf_fg h (hd (xs @ [y]))) ≤ flow_fg h y"
```

For this lemma it is crucial that the initial flow is at most the flow in the path's origin node. According to `inf_fg_le_flow_fg` this is the case for this lemma. In arrow notation we obtain (embezzling the assumptions):

$$i \xrightarrow{xs}_e y \leq f\ y$$

The next lemma relates the flow along a path to the inflow of a sibling flow graph:

```
lemma chain_inf_fg_le_inf_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "End_closed E" "∀x y. edge_fg h1 x y ∈ E" "set xs ⊆ dom_fg h1"
    "y ∈ dom_fg h2" "h = h1 + h2" "xs ≠ []" "id ∈ E" "(λ_. 0) ∈ E"
  shows "chain (edge_fg h1) xs y (inf_fg h1 (hd (xs @ [y]))) ≤ inf_fg h2 y"
```

In arrow notation we obtain:

$$i_1 \xrightarrow{xs}_{e_1} y \leq i_2\ y$$

Now we arrive at the next level of abstraction: capacities. The first lemma states that the entire capacity to a node y inside a flow graph can be approximated by y's flow:

```
lemma cap'_inf_fg_le_flow_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "∀x y. edge_fg h x y ∈ E" "End_closed E" "x ∈ dom_fg h"
    "y ∈ dom_fg h" "k ≥ 1" "id ∈ E" "(λ_. 0) ∈ E"
  shows "cap' k (dom_fg h) (edge_fg h) x y (inf_fg h x) ≤ flow_fg h y"
```

If we consider the capacity to a node outside the flow graph then we can approximate this capacity with the flow graph's outflow:

```
lemma cap_fg_inf_fg_le_outf_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "x ∈ dom_fg h" "y ∈ -dom_fg h" "End_closed E"
    "∀x y. edge_fg h x y ∈ E" "dom_fg h ≠ {}" "id ∈ E" "(λ_. 0) ∈ E"
  shows "cap_fg h x y (inf_fg h x) ≤ outf_fg h y"
```

From the previous lemma the next lemma follows directly by approximating the outflow with a sibling flow graphs's inflow:

```
lemma cap'_fg_inf_fg_fg_le_inf_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "h = h1 + h2" "xs ≠ []" "∀x y. edge_fg h x y ∈ E" "End_closed E"
    "∀x y. edge_fg h1 x y ∈ E" "∀x y. edge_fg h2 x y ∈ E" "id ∈ E" "(λ_. 0) ∈ E"
     "id ∈ E" "set xs ⊆ dom_fg h1" "y ∈ dom_fg h2"
   shows "cap_fg h1 (hd xs) y (inf_fg h1 (hd xs)) ≤ inf_fg h2 y"
```

The arrow notation of this lemma is

$$i_1 \xrightarrow{xs}\!\!\!\!\twoheadrightarrow_{e_1} y \leq i_2\ y$$

The next lemma demonstrates that the above lemmas can be generalized to initial flow values $m \leq \mathsf{inf}\ h\ x$ instead of fixed initial flow values like $\mathsf{inf}\ h\ x$:

```
lemma cap'_fg_le_inf_fg_le_inf_fg:
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "h ≠ bot" "h = h1 + h2" "xs ≠ []" "∀x y. edge_fg h x y ∈ E" "End_closed E"
    "∀x y. edge_fg h1 x y ∈ E" "∀x y. edge_fg h2 x y ∈ E" "id ∈ E"
    "set xs ⊆ dom_fg h1" "y ∈ dom_fg h2" "m ≤ inf_fg h1 (hd xs)" "(λ_. 0) ∈ E"
   shows "cap_fg h1 (hd xs) y m ≤ inf_fg h2 y"
```

The arrow notation of this lemma is

$$m \xrightarrow{xs}\!\!\!\!\rightarrow\!\!\!\!\rightarrow_e y \leq i_2\ y$$

The next level of abstraction is reached by approximating alternating capacity chains:

```
lemma chains'_cap_fg_inf_fg_le_inf_fg':
  fixes E :: "('a ⇒ 'a :: pos_cancel_comm_monoid_add) set"
  assumes "alternating' (P h1) (P h2) xss ys" "h ≠ bot" "h = h1 + h2" "xss ≠ []"
    "∀x y. edge_fg h x y ∈ E" "End_closed E" "∀x y. edge_fg h1 x y ∈ E"
    "∀x y. edge_fg h2 x y ∈ E" "xss ≠ []" "∀xs ∈ set xss. xs ≠ []"
    "⋀xs. P h1 xs ⟹ set xs ⊆ dom_fg h1" "⋀xs. P h2 xs ⟹ set xs ⊆ dom_fg h2"
    "id ∈ E" "hd ys ∈ dom_fg (alt h2 h1 xss)" "(λ_. 0) ∈ E"
  shows "chains' (alt_cap_fg h1 h2) xss (hd ys) (inf_fg h1 (hd (hd xss)))
    ≤ inf_fg (alt h2 h1 xss) (hd ys)"
```

This lemma considers a convoluted path through the subgraphs h_1 and h_2 of a flow graph $h = h_1 + h_2$. `alt_cap_fg` $\mathtt{h}_1$ $\mathtt{h}_2$ `xs y` is a function that simply chooses the appropriate capacity variant for path segments `xs`, i.e. it equals either `cap_fg` $\mathtt{h}_1$ (`hd xs`) `y` or `cap_fg` $\mathtt{h}_2$ (`hd xs`) `y` depending on `set xs` $\subseteq$ `dom_fg` $\mathtt{h}_1$ or not. The proof of this lemma applies symmetric alternating induction on `xss` and basically reduces to an application of `cap'_fg_inf_fg_le_inf_fg` for each path segment of `xss`.

6.3. Sourced Paths

In theorem 3 we have to come up with an arbitrary path xs within a flow graph $h' = h'_1 + h'_2$ that is sourced by inflow of h'.

In general, for effectively acyclic graphs h we can easily show (by sufficiently far unrolling of the flow equation and eliminating all terms still mentioning the flow using effective acyclicity as the pigeonhole principle tells us that there must be a cycle for long enough paths) that for every node x with flow $f\ x \neq 0$ there must be some path xs such that $i \xrightarrow{xs}_e x \neq 0$ (see `eff_acyclic_flow_is_sourced`).

Unfortunately, our flow graph $h' = h'_1 + h'_2$ is known to not be effectively acyclic. Fortunately, we know that there is an effectively acyclic flow graph $h = h_1 + h_2$ that is related to h' by subflow-preserving extensions $h_1 \preccurlyeq_S h'_1$ and $h_2 \preccurlyeq_S h'_2$. We additionally know that the interfaces of the corresponding flow graphs are equal, i.e. $i = i'$ and $i_k = i'_k$.

The proof idea is to obtain a cycle xs such that $f' \xrightarrow{xs}_{e'} \mathsf{hd}\, xs \neq 0$ as h' is not effectively acyclic. Then we start from node $x_0 \coloneqq \mathsf{hd}\, xs \in h'_k$ (see Figure 6.1) and find some h'_k-local sourced path $xs'_0 \subseteq h'_k$, i.e. $i'_k\ (\mathsf{hd}\, xs'_0) \neq 0$, using `eff_acyclic_flow_is_sourced` for h'_k. According to the definition of h'_k's inflow $i'_k\ (\mathsf{hd}\, xs'_0) = i'\ (\mathsf{hd}\, xs'_0) + o'_{3-k}\ (\mathsf{hd}\, xs'_0)$ there are two possibilities: there is inflow to h'_k contributed by i' or there is not. In the first case, we are done with our search for inflow. In the second case, we know that the entire inflow to $\mathsf{hd}\, xs'_0$ is contributed by h'_{3-k}.

From now on we continue the search for a sourced path in $h = h_1 + h_2$ instead of h'. We follow the inflow to $\mathsf{hd}\, xs'_0$ to a node in h_{3-k}. This is possible due to $o_k = o'_k$. Then we find some h_{3-k}-local sourced path $xs'_1 \subseteq h_{3-k}$. We again apply the case distinction whether the inflow to xs'_1 stems from i or i_k. In the first case we are done with out search, otherwise we do another step. We eventually find some convoluted alternating path $xss' = [xs'_1, \ldots, xs'_n]$ through h_1 and h_2, respectively, that is sourced by inflow to h. We now can transfer each path segment xs'_l for $l = 1, \ldots, n$ iteratively from h_k to h'_k using our subflow-preserving extensions $h_k \preccurlyeq_S h'_k$ and obtain a convoluted path $xss = [xs'_0, xs_1, \ldots, xs_n]$ in h'_1 and h'_2 that is sourced by inflow to h (and because of $i = i'$ also by inflow to h').

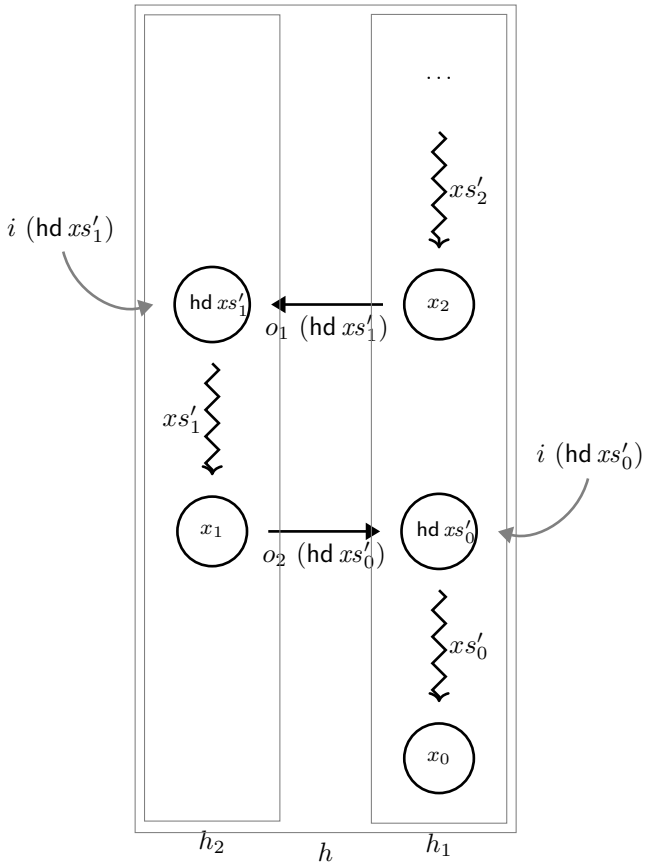


Figure 6.1.: Idea behind steps in lemma 1: We start at some node x_k and find a sourced path xs_k in x_k's subgraph. If $i\ (\mathsf{hd}\ xs_k) \neq 0$ we found what we are looking for. Otherwise, we find some node x_{k+1} in the other subgraph and repeat the same process once more.

We have to split this process into two lemmas: the first one finding a suitable starting node for the second lemma, and the second one executing the iterative steps starting from the suitable starting node. We start with the lemma on iterative steps:

Lemma 1. *Given flow graphs $h = h_1 + h_2$ and $h' = h'_1 + h'_2$ with $N_k = N'_k$ and $\mathsf{int}\, h_k = \mathsf{int}\, i'_k$ and $\mathsf{int}\, i = \mathsf{int}\, i'$ for $k = 1, 2$. Assume that h, h_1, h_2, h'_1, h'_2 are effectively acyclic and that h' is not effectively acyclic. Furthermore,*

$$\forall n \in h_k, n' \in -h'_k, m \leq i_k\ n.\ \mathsf{cap}\, h_k\ n\ n'\ m = \mathsf{cap}\, h'_k\ n\ n'\ m$$

and

$$o_2 \xrightarrow{xs}_e \xrightarrow{as}_{e'} b \neq 0$$

with $\mathsf{hd}\, xs \in h_1$, $xs \subseteq h$, $as \subseteq h'$. Then there exists a path $ys \subseteq h'$ such that

$$i \xrightarrow{ys}_{e'} \xrightarrow{xs}_e \xrightarrow{as}_{e'} b \neq 0$$

Proof. We conduct the proof by induction over $|xs|$ for arbitrary $h, h', h_1, h'_1, h_2, h'_2$ using the induction rule `nat_descend_induct`:

$$\dfrac{\bigwedge \mathtt{k}.\ \dfrac{\mathtt{n} < \mathtt{k}}{\mathtt{P\ k}} \qquad \bigwedge \mathtt{k}.\ \dfrac{\mathtt{k} \leq \mathtt{n} \qquad \bigwedge \mathtt{i}.\ \dfrac{\mathtt{k} < \mathtt{i}}{\mathtt{P\ i}}}{\mathtt{P\ k}}}{\mathtt{P\ m}}$$

The base case requires us to show that for paths xs with $|xs| > n$ the proposition holds in general. The step case requires us to show that for paths xs with $|xs| \leq n$ the proposition holds under the assumption that the proposition already holds for all longer paths. For our proof we instantiate the induction rule with some $n \geq \max\{|h|, |h'|\}$.

In parallel to the actual proposition of lemma 1 we have to prove an auxiliary proposition that provides us the pre-condition for the transfer of capacities using $\mathsf{cap}\, h_k\ n\ n'\ m = \mathsf{cap}\, h'_k\ n\ n'\ m$:

$$i \xrightarrow{ys}_{e'} (\mathsf{hd}\, xs) \leq i_1\ (\mathsf{hd}\, xs)$$

We now conduct the induction as stated:

- Base case $|xs| > \max\{|h|, |h'|\}$: Using the pigeonhole principle $|xs| > |h|$ and $xs \subseteq h$ imply that there is an element $x \in xs$ that occurs at least twice in xs. Therefore, we can decompose xs such that $xs = xs_1 \cdot x \cdot xs_2 \cdot x \cdot xs_3$.

  By assumptions we have

  $$o_2 \xrightarrow{xs}_e \xrightarrow{as}_e b \neq 0$$

  and obtain

  $$o_2 \xrightarrow{xs}_e (\mathsf{hd}\, (as \cdot [b])) \neq 0$$

  because our arrows are endomorphisms and therefore we can apply lemma $e_1\ (e_2\ x) \neq 0 \Longrightarrow e_2\ x \neq 0$. Substituting the decomposition of xs we obtain with another application of the lemma that

  $$o_2 \xrightarrow{xs_1}_e \xrightarrow{x \cdot xs_2}_e x \neq 0$$

  Using this we can derive the following contradiction:

  $$\begin{aligned}
  &0 < o_2 \xrightarrow{xs_1}_e \xrightarrow{x \cdot xs_2}_e x && \\
  &\leq f \xrightarrow{x \cdot xs_2}_e x && \text{by } \texttt{chain_inf_fg_le_flow_fg} \text{ and monotonicity} \\
  &= 0 && \text{by } h \text{ being effectively acyclic}
  \end{aligned}$$

  Therefore, this case can not occur.

- Descend Case $|xs| \leq \max\{|h|, |h'|\}$:

  We have

  $$\begin{aligned}
  &0 < o_2 \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by assumption} \\
  &= (\sum_{z \in h_2} f_2\ z \triangleright e_2\ z\ (\mathsf{hd}\, (xs \cdot as \cdot b)) \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by def. of outflow} \\
  &= (\sum_{z \in h_2} (\sum_{\substack{zs \subseteq h_2 \\ |zs| < |h_2|}} i_2 \xrightarrow{zs}_e z) \triangleright e_2\ z\ (\mathsf{hd}\, (xs \cdot as \cdot b))) \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by unrolling flow}
  \end{aligned}$$

$$
\begin{aligned}
&= (\sum_{z \in h_2} (\sum_{\substack{zs \subseteq h_2 \\ |zs| < |h_2|}} i_2 \xrightarrow{zs}_e z) \triangleright e\ z\ (\mathsf{hd}\,(xs \cdot as \cdot b))) \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by } z \in h_2 \\
&= (\sum_{z \in h_2} (\sum_{\substack{zs \subseteq h_2 \\ |zs| < |h_2|}} i_2 \xrightarrow{zs \cdot z}_e (\mathsf{hd}\,(xs \cdot as \cdot b)))) \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by } \texttt{chain\_append1} \\
&= (\sum_{z \in h_2} (\sum_{\substack{zs \subseteq h_2 \\ |zs| < |h_2|}} i_2 \xrightarrow{zs \cdot z \cdot xs}_e (\mathsf{hd}\,(as \cdot b)))) \xrightarrow{as}_{e'} b && \text{by endom.} \\
&= (\sum_{\substack{zs' \subseteq h_2 \\ 1 \leq |zs'| \leq |h_2|}} i_2 \xrightarrow{zs' \cdot xs}_e (\mathsf{hd}\,(as \cdot b))) \xrightarrow{as}_{e'} b && \text{by } zs' \coloneqq zs \cdot z \\
&= \sum_{\substack{zs' \subseteq h_2 \\ 1 \leq |zs'| \leq |h_2|}} i_2 \xrightarrow{zs' \cdot xs}_e \xrightarrow{as}_{e'} b && \text{by endom.}
\end{aligned}
$$

As the last sum does not equal 0 we obtain some $xs' \subseteq h_2$ with $xs' \neq []$ such that

$$i_2 \xrightarrow{xs' \cdot xs}_e \xrightarrow{as}_{e'} b \neq 0$$

We now show that there is some ys such that

$$i \xrightarrow{ys}_{e'} \xrightarrow{xs' \cdot xs}_e \xrightarrow{as}_{e'} b \neq 0$$

If $i \xrightarrow{xs' \cdot xs}_e \xrightarrow{as}_{e'} b \neq 0$ then we choose $ys = []$ and are done. Otherwise, it follows from the definition of sums of flow interfaces (in particular, $i_2 = i + o_1$) that

$$o_1 \xrightarrow{xs' \cdot xs}_e \xrightarrow{as}_{e'} b \neq 0$$

Using this we invoke the induction hypothesis for $h_1 \coloneqq h_2$ and $h_2 \coloneqq h_1$ and obtain some ys such that

$$i \xrightarrow{ys}_{e'} \xrightarrow{xs' \cdot xs}_e \xrightarrow{as}_{e'} b \neq 0$$

Additionally we obtain the auxiliary proposition

$$i \xrightarrow{ys}_{e'} (\mathsf{hd}\, xs') \leq i_2\ (\mathsf{hd}\, xs')$$

It remains to transfer xs' from h to h':

$$
\begin{aligned}
0 < i &\xrightarrow{ys}_{e'} \xrightarrow{xs' \cdot xs}_e \xrightarrow{as}_{e'} b \\
&= i \xrightarrow{ys}_{e'} \xrightarrow{xs'}_{e_2} \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by } xs' \subseteq h_2 \text{ and } h = h_1 + h_2 \\
&\leq i \xrightarrow{ys}_{e'} \xrightarrow{xs'}\!\!\!\!\twoheadrightarrow_{e_2} \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by } \texttt{chain\_le\_cap}' \\
&= i \xrightarrow{ys}_{e'} \xrightarrow{xs'}\!\!\!\!\twoheadrightarrow_{e_2'} \xrightarrow{xs}_e \xrightarrow{as}_{e'} b && \text{by assumption and aux. prop.}
\end{aligned}
$$

The last equality follows from our assumption that

$$\forall n \in h_2, n' \in -h_2', m \leq i_2\ n.\ \mathsf{cap}\, h_2\ n\ n'\ m = \mathsf{cap}\, h_2'\ n\ n'\ m$$

We know that $\mathsf{hd}\, xs' \in h_2$ and $\mathsf{hd}\, xs \in h_1 \subseteq -h_2$. For $ys \neq []$ the auxiliary proposition provided by the induction hypothesis ensures that $m \coloneqq i \xrightarrow{ys}_{e'} (\mathsf{hd}\, xs') \leq i_2\ (\mathsf{hd}\, xs')$. For $ys = []$ this assumption follows trivially from the definition of chain.

After unfolding the definition of cap we obtain some $ys' \subseteq h_2$ with $\mathsf{hd}\, ys' = \mathsf{hd}\, xs$ such that

$$\begin{aligned}
0 \neq i &\xrightarrow{ys}_{e'} \xrightarrow{ys'}_{e'_2} \xrightarrow{xs}_{e} \xrightarrow{as}_{e'} b \\
&\neq i \xrightarrow{ys}_{e'} \xrightarrow{ys'}_{e'} \xrightarrow{xs}_{e} \xrightarrow{as}_{e'} b && \text{by } ys' \subseteq h_2 \text{ and } h = h_1 + h_2 \\
&= i \xrightarrow{ys \cdot ys'}_{e'} \xrightarrow{xs}_{e} \xrightarrow{as}_{e'} b
\end{aligned}$$

For $ys \coloneqq ys \cdot ys'$ this is exactly what we had to show.

Now, we are only missing the proof for the auxiliary proposition:

$$\begin{aligned}
i' \xrightarrow{ys \cdot ys'}_{e'} \mathsf{hd}\, xs &= i' \xrightarrow{ys}_{e'} \xrightarrow{ys'}_{e'} \mathsf{hd}\, xs \\
&= i' \xrightarrow{ys}_{e'} \xrightarrow{ys'}_{e'_2} \mathsf{hd}\, xs && \text{by } ys' \subseteq h_2 \text{ and } h' = h'_1 + h'_2 \\
&\leq i'_2 \xrightarrow{ys'}_{e_2} \mathsf{hd}\, xs && \text{by aux. prop or } \texttt{inf_fg_le_inf_fg} \\
&\leq i'_1\ (\mathsf{hd}\, xs) && \text{by } \texttt{chain_inf_fg_le_inf_fg}
\end{aligned}$$

Note that the third step applies a case distinction whether $ys \neq []$: if true then we apply the auxiliary proposition after remembering that $\mathsf{hd}\, ys' = \mathsf{hd}\, xs'$, if false then the arrow equals id and we apply `inf_fg_le_inf_fg`.

■

The next lemma encapsulates the lemma on iterative search and provides it with some initial path such that we have some valid starting node for lemma 1:

Lemma 2. *Given flow graphs $h = h_1 + h_2$ and $h' = h'_1 + h'_2$ with $\mathsf{dom}\, h_k = \mathsf{dom}\, h'_k$ and $\mathsf{int}\, h_k = \mathsf{int}\, h'_k$ and $\mathsf{int}\, h = \mathsf{int}\, h'$ for $k = 1, 2$. Assume that h, h_1, h_2, h'_1, h'_2 are effectively acyclic and that h' is not effectively acyclic. Furthermore,*

$$\forall n \in h_k, n' \in -h'_k, m \leq i_k\ n.\ \mathsf{cap}\, h_k\ n\ n'\ m = \mathsf{cap}\, h'_k\ n\ n'\ m$$

Then there exists a path $xs \subseteq h'$ and some $y \in xs$ such that

$$i \xrightarrow{xs}_{e'} y \neq 0$$

Proof. By assumption we know that h' is not effectively acyclic and therefore obtain by definition of effective acyclicity some path as such that $f' \xrightarrow{as}_{e'} \mathsf{hd}\, as \neq 0$ and $as \subseteq h'$. Without loss of generality we assume that $\mathsf{hd}\, as \in h'_1$. Using `eff_acyclic_flow_is_sourced` we obtain some path $xs \subseteq h'_1$ such that

$$i'_1 \xrightarrow{xs}_{e'_1} \xrightarrow{as}_{e'} \mathsf{hd}\, as \neq 0$$

Because of $i'_1 = i' + o'_2$ we have to consider two cases:

- If $i' \xrightarrow{xs}_{e'_1} \xrightarrow{as}_{e'} (\mathsf{hd}\, as) \neq 0$ then we directly found a solution $xs \coloneqq xs \cdot as$ and $y \coloneqq \mathsf{hd}\, as$:

$$\begin{aligned}
i \xrightarrow{xs \cdot as}_{e'} (\mathsf{hd}\, as) &= i \xrightarrow{xs}_{e'} \xrightarrow{as}_{e'} \mathsf{hd}\, as \\
&= i \xrightarrow{xs}_{e'_1} \xrightarrow{as}_{e'} \mathsf{hd}\, as && \text{by } xs \subseteq h'_1 \\
&= i' \xrightarrow{xs}_{e'_1} \xrightarrow{as}_{e'} \mathsf{hd}\, as && \text{by } i' = i \\
&\neq 0
\end{aligned}$$

- Otherwise, $o_2' \xrightarrow{xs}_{e_1'} \xrightarrow{as}_{e'} \mathsf{hd}\ as \neq 0$. We obtain

$$\begin{aligned} o_2 \xrightarrow{xs \cdot as}_{e'} \mathsf{hd}\ as &= o_2' \xrightarrow{xs \cdot as}_{e'} \mathsf{hd}\ as && \text{by } o_2' = o_2 \\ &= o_2' \xrightarrow{xs}_{e'} \xrightarrow{as}_{e'} \mathsf{hd}\ as \\ &= o_2' \xrightarrow{xs}_{e_1'} \xrightarrow{as}_{e'} \mathsf{hd}\ as && \text{by } xs \subseteq h_1' \\ &\neq 0 \end{aligned}$$

  Using this we can apply lemma 1 for $b := \mathsf{hd}\ as$, $as := xs \cdot as$, $xs := []$ and obtain some ys with $ys \subseteq h'$ such that

$$i \xrightarrow{ys \cdot xs \cdot as}_{e'} \mathsf{hd}\ as \neq 0$$

  Therefore, we found the solution $xs := ys \cdot xs \cdot as$ and $y := \mathsf{hd}\ as$.

■

6.4. Transfer of Capacity Chains

lemma 1 already exhibited the transfer of capacities from a flow graph h to a flow graph h' using subflow-preserving extensions. However, in lemma 1 we only considered a single capacity between two nodes because we had to select a single path after each induction step. We now consider the transfer of alternating capacity chains from $h = h_1 + h_2$ to $h' = h_1' + h_2'$.

Lemma 3. *Let $h = h_1 + h_2 \neq \bot$ and $h' = h_1' + h_2 \neq \bot$ be flow graphs with $\mathsf{dom}\, h_k = \mathsf{dom}\, h_k'$ for $k = 1, 2$. Let $P = \lambda h\ xs.\ xs \neq [] \wedge xs \subseteq h$ and*

$$F = \lambda h_1\ h_2\ xs\ y.\ \begin{cases} \mathsf{cap}\, h_1\ (\mathsf{hd}\, xs)\ y & \text{for } xs \subseteq h_1 \\ \mathsf{cap}\, h_2\ (\mathsf{hd}\, xs)\ y & \text{for } xs \subseteq h_2 \\ 0 & \text{otherwise} \end{cases}$$

Let xss be a convoluted list with $\mathsf{alternating}'\ (P\ h_1)\ (P\ h_2)\ xss\ ys$. Furthermore,

$$\forall n \in h_k, n' \in -h_k', m \leq i_k\ n.\ \mathsf{cap}\, h_k\ n\ n'\ m = \mathsf{cap}\, h_k'\ n\ n'\ m$$

and $m \leq i_1(\mathsf{hd}\,(\mathsf{hd}\, xss))$. Then

$$m \xrightarrow{xss}\!\!\!\!\rightarrow_{F\ h_1\ h_2} (\mathsf{hd}\, ys) = m \xrightarrow{xss}\!\!\!\!\rightarrow_{F\ h_1'\ h_2'} (\mathsf{hd}\, ys)$$

Proof. By symmetric alternating induction over xss for arbitrary m.

- The empty case $xss = []$ is trivial:

$$m \xrightarrow{[]}\!\!\!\!\rightarrow_{F\ h_1\ h_2} (\mathsf{hd}\, ys) = m = m \xrightarrow{[]}\!\!\!\!\rightarrow_{F\ h_1'\ h_2'} (\mathsf{hd}\, ys)$$

- The base case $xss = [xs]$ with $P\ h_1\ xs$ and $P\ h_2\ ys$ reduces to the assumed equality between capacities:

$$\begin{aligned} m \xrightarrow{xs}\!\!\!\!\rightarrow_{F\ h_1\ h_2} \mathsf{hd}\, ys &= \mathsf{cap}\, h_1\ (\mathsf{hd}\, xs)\ (\mathsf{hd}\, ys)\ m \\ &= \mathsf{cap}\, h_1'\ (\mathsf{hd}\, xs)\ (\mathsf{hd}\, ys)\ m \\ &= m \xrightarrow{xs}\!\!\!\!\rightarrow_{F\ h_1'\ h_2'} \mathsf{hd}\, ys \end{aligned}$$

  We can apply the equality between capacities because $\mathsf{alternating}\,(P\ h_1)\ (P\ h_2)\ [xs]\ ys$ implies $\mathsf{hd}\, xs \in h_1$ and $\mathsf{hd}\, ys \in h_2$ and by assumption $m \leq i_1(\mathsf{hd}\, xs)$.

- The step case $xss = xs_1 \cdot xs_2 \cdot xss'$ provides us with $P\ h_1\ xs_1$ and $P\ h_2\ xs_2$ and the induction hypothesis for $xs_2 \cdot xss'$. We then calculate

$$\begin{aligned} m \xrightarrow{xs_1 \cdot xs_2 \cdot xss'}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1\ h_2} \mathsf{hd}\ ys &= m \xrightarrow{xs_1}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1\ h_2} \xrightarrow{xs_2 \cdot xss'}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1\ h_2} \mathsf{hd}\ ys \\ &= m \xrightarrow{xs_1}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1\ h_2} \xrightarrow{xs_2 \cdot xss'}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1'\ h_2'} \mathsf{hd}\ ys \\ &= m \xrightarrow{xs_1}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1'\ h_2'} \xrightarrow{xs_2 \cdot xss'}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1'\ h_2'} \mathsf{hd}\ ys \\ &= m \xrightarrow{xs_1 \cdot xs_2 \cdot xss'}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1'\ h_2'} \mathsf{hd}\ ys \end{aligned}$$

  The first and last equality follow from the definition of chains'. The second equality follows from the induction hypothesis after noting that $m \xrightarrow{xs_1}\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{F\ h_1\ h_2} \mathsf{hd}\ xs_2 \leq \mathsf{inf}\ h_2\ (\mathsf{hd}\ xs_2)$ by `cap'_inf_fg_le_inf_fg` and $\mathsf{alternating}'\ (P\ h_2)\ (P\ h_1)\ (xs_2 \cdot xss')\ ys$. The third equality follows analogously to the transfer of capacities in the base case.

■

We will also use a slight variation of lemma 3 that works on sequences of common nodes instead of approximated paths:

Lemma 4. *Let $h = h_1 + h_2 \neq \bot$ and $h' = h_1' + h_2 \neq \bot$ be flow graphs with $\mathsf{dom}\ h_k = \mathsf{dom}\ h_k'$ for $k = 1, 2$. Let $P = \lambda h\ x.\ x \in h$ and*

$$F = \lambda h_1\ h_2\ x\ y. \begin{cases} \mathsf{cap}\ h_1\ x\ y & \text{for } xs \in h_1 \\ \mathsf{cap}\ h_2\ x\ y & \text{for } xs \in h_2 \\ 0 & \text{otherwise} \end{cases}$$

Let xs be a list of common nodes with $\mathsf{alternating}\ (P\ h_1)\ (P\ h_2)\ xs\ y$ with $y \in -h$. Furthermore,

$$\forall n \in h_k, n' \in -h_k', m \leq i_k\ n.\ \mathsf{cap}\ h_k\ n\ n'\ m = \mathsf{cap}\ h_k'\ n\ n'\ m$$

and $m \leq i_1(\mathsf{hd}\ (\mathsf{hd}\ xss))$. Then

$$\mathsf{chain}\ (F\ h_1\ h_2)\ xs\ y\ m = \mathsf{chain}\ (F\ h_1'\ h_2'\ )\ xs\ y\ m$$

6.5. Transfer of Capacities

The next level of transferring paths between flow graphs related by subflow-preserving extensions is to transfer a capacity from a sum flow-graph $h = h_1 + h_2$ to $h' = h_1' + h_2'$. The previous section considered chains of capacities where we were able to transfer a single chain segment at a time. In this section we have to first find a decomposition of capacities in h into chains of capacities in h_1 and h_2. Only then we can apply the previous result to obtain the required transfer of a capacity.

To decompose a capacity in h into capacity chains in h_1 and h_2 we prove the following lemma:

Lemma 5. *Let $h = h_1 + h_2 \neq \bot$ be an effectively acyclic flow graph and*

$$F = \lambda h_1\ h_2\ x\ y. \begin{cases} \mathsf{cap}\ h_1\ x\ y & \text{for } x \in h_1 \\ \mathsf{cap}\ h_2\ x\ y & \text{for } x \in h_2 \\ 0 & \text{otherwise} \end{cases}$$

Then for all $x \in h_1$, $x' \in -h$ and $m \leq \mathsf{inf}\ h\ x$ holds that

$$\mathsf{cap}\ h\ x\ x'\ m = \sum_{k=1}^{|h|} \sum_{\substack{\mathsf{length}\ xs = k \\ \mathsf{distinct}\ xs \\ \mathsf{hd}\ xs = x \\ \mathsf{alternating}\ h_1\ h_2\ xs}} \mathsf{chain}\ (F\ h_1\ h_2)\ xs\ x'\ m$$

Proof. We omit the correctness arguments of the sum transformations in this paper proof. Please refer to our formalization for these.

We calculate:

$$\begin{aligned}
\mathsf{cap}\ h\ x\ x'\ m &= \sum_{\substack{x \cdot xs \subseteq h \\ |x \cdot xs| \leq |h| \\ \mathsf{distinct}\ (x \cdot xs)}} \mathsf{chain}\ e\ (x \cdot xs)\ x'\ m \\
&= \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{hd}\ (\mathsf{hd}\ xss) = x \\ \mathsf{alternating}\ h_1\ h_2\ xss}} \mathsf{chain}\ e\ (\mathsf{concat}\ xss)\ x'\ m \\
&= \sum_{k=1}^{|h|} \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{hd}\ (\mathsf{hd}\ xss) = x \\ \mathsf{alternating}\ h_1\ h_2\ xss \\ \mathsf{length}\ xss = k}} \mathsf{chain}\ e\ (\mathsf{concat}\ xss)\ x'\ m \\
&= \sum_{k=1}^{|h|} \sum_{\substack{\mathsf{length}\ ys = k \\ \mathsf{distinct}\ ys \\ \mathsf{hd}\ ys = x \\ \mathsf{alternating}\ h_1\ h_2\ ys}} \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map}\ \mathsf{hd}\ xss = ys \\ \mathsf{alternating}\ h_1\ h_2\ xss}} \mathsf{chain}\ e\ (\mathsf{concat}\ xss)\ x'\ m \\
&= \sum_{k=1}^{|h|} \sum_{\substack{\mathsf{length}\ ys = k \\ \mathsf{distinct}\ ys \\ \mathsf{hd}\ ys = x \\ \mathsf{alternating}\ h_1\ h_2\ ys}} \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map}\ \mathsf{hd}\ xss = ys \\ \mathsf{alternating}\ h_1\ h_2\ xss \\ \mathsf{alternating}\ (|\cdot| \leq |h_1|)\ (|\cdot| \leq |h_2|)\ xss}} \mathsf{chain}\ e\ (\mathsf{concat}\ xss)\ x'\ m \\
&= \sum_{k=1}^{|h|} \sum_{\substack{\mathsf{length}\ ys = k \\ \mathsf{distinct}\ ys \\ \mathsf{hd}\ ys = x \\ \mathsf{alternating}\ h_1\ h_2\ ys}} \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map}\ \mathsf{hd}\ xss = ys \\ \mathsf{alternating}\ h_1\ h_2\ xss \\ \mathsf{alternating}\ (|\cdot| \leq |h_1|)\ (|\cdot| \leq |h_2|)\ xss}} \mathsf{chains}\ e\ xss\ x'\ m \\
&= \sum_{k=1}^{|h|} \sum_{\substack{\mathsf{length}\ ys = k \\ \mathsf{distinct}\ ys \\ \mathsf{hd}\ ys = x \\ \mathsf{alternating}\ h_1\ h_2\ ys}} \mathsf{chain}\ (F\ h_1\ h_2)\ ys\ x'\ m
\end{aligned}$$

The first step follows from unrolling the capacity and noting that all summands with $\neg\mathsf{distinct}\,(x \cdot xs)$ equal 0 due to effective acyclicity of h. We explicitly record that $|x \cdot xs| \leq |h|$ which is already implied by $x \cdot xs \subseteq h$ and $\mathsf{distinct}\,(x \cdot xs)$. The second step simply rewrites the sum variable of the big-sum: we always can find a decomposition of $x \cdot xs$ into an alternating list xss of non-empty lists using `split_segments`. In the third step we partition the big-sum according to the length of xss. The fourth step partitions the inner big-sums according to the head elements of the lists xss. The fifth step prunes summands using effective acyclicity of h_1 and h_2, i.e. if there is a path segment $xs_l \subseteq h_k$ in some xss with $|xs_l| > |h_k|$ then this path segment has no flow and, therefore, the entire path has not. The sixth step transitions from chain to chains. Now, if the inner big-sum corresponds to $\mathsf{chain}\,(F\,h_1\,h_2)\ ys\ x'\ m$ then we finished the proof. We show this equality by symmetric alternating induction on ys for arbitrary m:

- Empty case $ys = []$: does not occur due to $k \geq 1$ and $\mathsf{length}\,ys = k$.
- Base case $ys = [y_0]$: by exploiting this information the goal reduces to

$$\mathsf{cap}\,h_1\ y_0\ x'\ m = \sum_{\substack{xs \subseteq h_1 \\ |xs| \leq |h_1| \\ \mathsf{distinct}\ xs \\ \mathsf{hd}\ xs = y_0}} \mathsf{chain}\,e\ xs\ x'\ m$$

  We argue analogously to the first two steps of the previous part of the proof: after unrolling the capacity on the left hand side, eliminating all summands for non-distinct xs that evaluate to 0 due to effective acyclicity of h_1, and integrating y_0 into the sum variable we obtain exactly the right hand side.

- Step case $ys = y_0 \cdot y_1 \cdot ys'$:

$$\begin{aligned}
&\mathsf{chain}\,(F\,h_1\,h_2)\ (y_0 \cdot y_1 \cdot ys)\ x'\ m \\
&= \mathsf{chain}\,(F\,h_1\,h_2)\ (y_1 \cdot ys)\ x'\ (\mathsf{cap}\,h_1\ y_0\ y_1\ m) \\
&= \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map\ hd}\ xss = y_1 \cdot ys \\ \mathsf{alternating}\ h_2\ h_1\ xss \\ \mathsf{alternating}\ (|\cdot| \leq |h_2|)\ (|\cdot| \leq |h_1|)\ xss}} \mathsf{chains}\,e\ xss\ x'\ (\mathsf{cap}\,h_1\ y_0\ y_1\ m) \\
&= \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map\ hd}\ xss = y_1 \cdot ys \\ \mathsf{alternating}\ h_2\ h_1\ xss \\ \mathsf{alternating}\ (|\cdot| \leq |h_2|)\ (|\cdot| \leq |h_1|)\ xss}} \mathsf{chains}\,e\ xss\ x'\ \Big(\sum_{\substack{y_0 \cdot zs \subseteq h_1 \\ |y_0 \cdot zs| \leq |h_1| \\ \mathsf{distinct}\ (y_0 \cdot zs)}} \mathsf{chain}\,e_1\ (y_0 \cdot zs)\ y_1\ m\Big) \\
&= \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map\ hd}\ xss = y_1 \cdot ys \\ \mathsf{alternating}\ h_2\ h_1\ xss \\ \mathsf{alternating}\ (|\cdot| \leq |h_2|)\ (|\cdot| \leq |h_1|)\ xss}} \sum_{\substack{y_0 \cdot zs \subseteq h_1 \\ |y_0 \cdot zs| \leq |h_1| \\ \mathsf{distinct}\ ((y_0 \cdot zs) \cdot \mathsf{concat}\ xss)}} \mathsf{chains}\,e\ xss\ x'\ (\mathsf{chain}\,e_1\ (y_0 \cdot zs)\ y_1\ m) \\
&= \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map\ hd}\ xss = y_1 \cdot ys \\ \mathsf{alternating}\ h_2\ h_1\ xss \\ \mathsf{alternating}\ (|\cdot| \leq |h_2|)\ (|\cdot| \leq |h_1|)\ xss}} \sum_{\substack{y_0 \cdot zs \subseteq h_1 \\ |y_0 \cdot zs| \leq |h_1| \\ \mathsf{distinct}\ ((y_0 \cdot zs) \cdot \mathsf{concat}\ xss)}} \mathsf{chains}\,e\ ((y_0 \cdot zs) \cdot xss)\ x'\ m
\end{aligned}$$

$$= \sum_{\substack{\mathsf{concat}\ xss \subseteq h \\ |\mathsf{concat}\ xss| \leq |h| \\ \mathsf{distinct}\ (\mathsf{concat}\ xss) \\ \mathsf{map}\ \mathsf{hd}\ xss = y_0 \cdot y_1 \cdot ys \\ \mathsf{alternating}\ h_1\ h_2\ xss \\ \mathsf{alternating}\ (|\cdot| \leq |h_1|)\ (|\cdot| \leq |h_2|)\ xss}} \mathsf{chains}\, e\ ((y_0 \cdot zs) \cdot xss)\ x'\ m$$

The first step simply unfolds the definition of chain. The second step applies the induction hypothesis. Step three unrolls $\mathsf{cap}\, h_1\ y_0\ y_1\ m$. In step four we partition the inner big-sum into those zs such that $\mathsf{distinct}\, ((y_0 \cdot zs) \cdot \mathsf{concat}\, xss)$ holds and those zs that do not satisfy this property. Then we distribute $\mathsf{chains}\, e\ xss\ x'$ into both partition big-sums and note that the second partition vanishes due to effective acyclicity. In step five we join chain and chains according to the definition of chains after noticing that $\mathsf{hd}\, (\mathsf{hd}\, xss) = y_1$. In step six we integrate $y_0 \cdot zs$ into zss and combine both big-sums and are done.

■

Using lemma 5 we can reduce the transfer of capacities to the transfer of capacity chains as provided by lemma 4:

Lemma 6. *Let $h = h_1 + h_2 \neq \bot$ and $h' = h'_1 + h_2 \neq \bot$ be flow graphs with $\mathsf{dom}\, h_k = \mathsf{dom}\, h'_k$ for $k = 1, 2$. Let $F = \lambda h\ xs.\ xs \neq [] \wedge xs \subseteq h$. Let $n \in h$ and $n' \in h'$. Assume that*

$$\forall n \in h_k, n' \in -h'_k, m \leq i_k\ n.\ \mathsf{cap}\, h_k\ n\ n'\ m = \mathsf{cap}\, h'_k\ n\ n'\ m$$

and $m \leq i\ n$. Then

$$\mathsf{cap}\, h\ n\ n'\ m = \mathsf{cap}\, h'\ n\ n'\ m$$

Proof. We assume without loss of generality that $n \in h_1$. Otherwise exchange h_1 and h_2. Then

$$\begin{aligned}
\mathsf{cap}\, h\ n\ n'\ m &= \sum_{k=1}^{|h|} \sum_{\substack{\mathsf{length}\ xs = k \\ \mathsf{distinct}\ xs \\ \mathsf{hd}\ xs = x \\ \mathsf{alternating}\ h_1\ h_2\ xs}} \mathsf{chain}\, (F\, h_1\, h_2)\ xs\ x'\ m \\
&= \sum_{k=1}^{|h'|} \sum_{\substack{\mathsf{length}\ xs = k \\ \mathsf{distinct}\ xs \\ \mathsf{hd}\ xs = x \\ \mathsf{alternating}\ h'_1\ h'_2\ xs}} \mathsf{chain}\, (F\, h'_1\, h'_2)\ xs\ x'\ m \\
&= \mathsf{cap}\, h'\ n\ n'\ m
\end{aligned}$$

The first and last equality are justified by lemma 5, the second equality is justified for each k and xs by lemma 4 and by noting that $\mathsf{dom}\, h_k = \mathsf{dom}\, h'_k$ and thereby also $|h| = |h'|$ using $h = h_1 + h_2$. ■

6.6. Three More Lemmas

In the following we provide proofs for lemmas 3.39 to 3.41 from [Krishna, 2019].

Lemma 7. *If $h' = h_0 + h \neq \bot$ such that $\forall n\, n'.\ \mathsf{edge}\, h_0\ n\ n' = 0$ and h is effectively acyclic, then h' is effectively acyclic.*

Proof. To show effective acyclicity we assume some $k \geq 1$ and a list ns such that $|ns| = k$ and $ns \subseteq h'$. We have to show that $f' \xrightarrow{ns}_{e'} \mathsf{hd}\ ns = 0$.

If $ns \subseteq h$ then effective acyclicity of h provides this statement because edges and flows of subgraphs are identical to the corresponding parts of h'. Therefore, we can assume that there is some element n in ns such that $n \in h_0$. Using $n \in h_0$ we obtain a decomposition $ns = ns_1 \cdot n \cdot ns_2$. Then

$$
\begin{aligned}
f' \xrightarrow{ns}_{e'} \mathsf{hd}\ ns &= f' \xrightarrow{ns_1 \cdot n \cdot ns_2}_{e'} \mathsf{hd}\ ns && \text{by decomposition} \\
&= (f' \xrightarrow{ns_1}_{e'} n) \xrightarrow{[n]}_{e'} \xrightarrow{ns_2}_{e'} \mathsf{hd}\ ns && \text{by arrow calculus} \\
&= ((f' \xrightarrow{ns_1}_{e'} n) \triangleright e'\ n\ (\mathsf{hd}\ ns)) \xrightarrow{ns_2}_{e'} \mathsf{hd}\ ns && \text{by def. of arrow} \\
&= ((f' \xrightarrow{ns_1}_{e'} n) \triangleright e_0\ n\ (\mathsf{hd}\ ns)) \xrightarrow{ns_2}_{e'} \mathsf{hd}\ ns && \text{by } n \in h_0 \text{ and } h' = h_0 + h \neq \bot \\
&= 0 \xrightarrow{ns_2}_{e'} \mathsf{hd}\ ns && \text{by } n \in h_0 \text{ and } e_0\ n\ \_ = 0 \\
&= 0 && \text{by endomorphism}
\end{aligned}
$$

■

Lemma 8. *If $h' = h_0 + h \neq \bot$ such that $\forall n\, n'.\ \mathsf{edge}\, h_0\ n\ n' = 0$, then*

$$\forall n' \in -h'.\ \mathsf{outf}\ h'\ n = \mathsf{outf}\ h\ n$$

Proof. Assume $n' \in -h'$. We have to show $\mathsf{outf}\ h'\ n = \mathsf{outf}\ h\ n$.

$$
\begin{aligned}
\mathsf{outf}\ h'\ n &= \sum_{n' \in h'} f'\ n' \triangleright e'\ n'\ n && \text{by def. of outflow} \\
&= (\sum_{n' \in h_0} f'\ n' \triangleright e'\ n'\ n) + (\sum_{n' \in h} f'\ n' \triangleright e'\ n'\ n) && \text{by } h' = h_0 + h \neq \bot \\
&= (\sum_{n' \in h_0} f_0\ n' \triangleright e_0\ n'\ n) + (\sum_{n' \in h} f\ n' \triangleright e\ n'\ n) && \text{by def. of } + \text{ for flow graphs} \\
&= (\sum_{n' \in h} f\ n' \triangleright e\ n'\ n) && \text{by assumption } e_0\ n'\ \_ = 0 \text{ for } n' \in h_0 \\
&= \mathsf{outf}\ h\ n && \text{by def. of outflow}
\end{aligned}
$$

■

Lemma 9. *If $h' = h_0 + h \neq \bot$ such that $\forall n\, n'.\ \mathsf{edge}\, h_0\ n\ n' = 0$ and h is effectively acyclic, then*

$$\forall n \in h',\ n' \in -h'.\ m \leq \mathsf{inf}\ h'\ m \longrightarrow m \triangleright \mathsf{cap}\, h'\ n\ n' = \begin{cases} m \triangleright \mathsf{cap}\, h\ n\ n' & \text{for } n \in h \\ 0 & \text{for } n \in h_0 \end{cases}$$

Proof. We assume that $n \in h'$, $n' \in -h'$ and that $m \leq \mathsf{inf}\ h'\ m$. We have to show that

$$m \triangleright \mathsf{cap}\, h'\ n\ n' = \begin{cases} m \triangleright \mathsf{cap}\, h\ n\ n' & \text{for } n \in h \\ 0 & \text{for } n \in h_0 \end{cases}$$

If $n \in h$ then

$$\mathsf{cap}\ \ h'\ n\ n'\ m = \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'|}} m \xrightarrow{n \cdot ns}_{e'} n')$$

$$
\begin{aligned}
&= \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'| \\ ns \subseteq h}} m \xrightarrow{n \cdot ns}_{e'} n') + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'| \\ ns \not\subseteq h}} m \xrightarrow{n \cdot ns}_{e'} n') \\
&= \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'| \\ ns \subseteq h}} m \xrightarrow{n \cdot ns}_{e'} n') \\
&= \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'| \\ ns \subseteq h}} m \xrightarrow{n \cdot ns}_{e} n') \\
&= \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'| \\ ns \subseteq h \\ |ns| < |h|}} m \xrightarrow{n \cdot ns}_{e} n') + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'| \\ ns \subseteq h \\ |ns| \geq |h|}} m \xrightarrow{n \cdot ns}_{e} n') \\
&= \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'| \\ ns \subseteq h \\ |ns| < |h|}} m \xrightarrow{n \cdot ns}_{e} n') \\
&= \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h \\ |ns| < |h|}} m \xrightarrow{n \cdot ns}_{e} n') \\
&= \mathsf{cap}\, h\ n\ n'\ m
\end{aligned}
$$

The first step simply unrolls the capacity in h'. The second step merely partitions the big-sum: is the entire path contained in h? The third step notices that for each $ns \not\subseteq h$ there is some element n_0 in ns such that $n_0 \in h_0$. Therefore, $m \xrightarrow{n \cdot ns}_{e'} n' = 0$ analogously to the reasoning in lemma 7. The fourth step transfers the paths from h' to h as all $ns \subseteq h$ and $h' = h_0 + h$. The fifth step again partitions the big-sum according to the length of the paths. The sixth step notices that effective acyclicity of h implies for all ns with $|ns| \geq |h|$ that $m \xrightarrow{n \cdot ns}_{e} n' = 0$. The seventh step only omits the redundant conditions on the sum (due to $\mathsf{dom}\, h \subseteq \mathsf{dom}\, h'$ and $|h| \leq |h'|$), such that the last step can un-unroll the capacity in h and finished this case.

For $n \notin h$ it follows that $n \in h_0$. Then

$$
\mathsf{cap}\ \ h'\ n\ n'\ m = \delta\ n\ n'\ m + (\sum_{\substack{ns \subseteq h' \\ |ns| < |h'|}} m \xrightarrow{n \cdot ns}_{e'} n') = \delta\ n\ n'\ m = 0
$$

The first step again unrolls the capacity in h'. The second step notices analogously to the third step of the previous case that the big sum equals 0, this time due to $n \in h_0$. The last step notices $\delta\ n\ n'\ m = \texttt{if}\ n = n'\ \texttt{then}\ m\ \texttt{else}\ 0 = 0$ as $n \in \mathsf{dom}\, h_0 \subseteq \mathsf{dom}\, h'$ and $n' \notin h'$. ■

6.7. Proof of [Krishna, 2019, Theorem 3]

We finally prove [Krishna, 2019, th. 3]:

Theorem 1. *Let M be a positive monoid, E be a point-wise reduced set of endomorphisms. If $h = h_1 + h_2$, $h_1 \preccurlyeq_S h_1'$, $h_1' \cap h_2 = \emptyset$, $\forall n \in h_1' - h_1$. $\mathsf{outf}\, h_2\ n = 0$, and if h, h_1, h_2, h_1' are effectively acyclic flow graphs, then there exists an effectively acyclic flow graph $h' = h_1' + h_2$ such that $h \preccurlyeq_S h'$.*

This proof basically follows the proof already given in [Krishna, 2019] but refines the proof significantly. The proof is structured as follows: first, we consider the case where $\mathsf{dom}\, h_1 = \mathsf{dom}\, h_2$, and second, we consider the case where $\mathsf{dom}\, h_1 \subsetneq \mathsf{dom}\, h_2$ by reducing it to the first one.

Proof. **Case 1:** $\mathsf{dom}\, h_1 = \mathsf{dom}\, h_2$

The case distinction provides us that $\mathsf{dom}\, h_1 = \mathsf{dom}\, h_2$. This assumption reduces our assumption on the subflow-preserving extension $h_1 \preccurlyeq_S h_1'$ to $\mathsf{int}\, h_1 = \mathsf{int}\, h_1'$ and

$$\forall n \in h_1, n' \in -h_1', m \leq \mathsf{inf}\, h_1\ n.\ m \triangleright \mathsf{cap}\, h_1\ n\ n' = m \triangleright \mathsf{cap}\, h_1'\ n\ n' \tag{6.1}$$

The homomorphic relationship between flow graphs and interfaces (`int_fg_fi_hom`) then implies together with $\bot \neq h = h_1 + h_2$ and $h' = h_1' + h_2$ that

$$\bot \neq \mathsf{int}\, h = \mathsf{int}\, h_1 + \mathsf{int}\, h_2 = \mathsf{int}\, h_1' + \mathsf{int}\, h_2 = \mathsf{int}\, h'$$

It remains to show that h' is effectively acyclic and $h \preccurlyeq_S h'$. For notational reasons we introduce a variable $h_2' \coloneqq h_2$ for the remainder of this proof and note that

$$\forall n \in h_2, n' \in -h_2', m \leq \mathsf{inf}\, h_2\ n.\ m \triangleright \mathsf{cap}\, h_2\ n\ n' = m \triangleright \mathsf{cap}\, h_2'\ n\ n' \tag{6.2}$$

trivially holds.

Case 1, EA

First, we show that h' is effectively acyclic. We do this by proof by contradiction, i.e. we assume that h' is not effectively acyclic and have to infer a contradiction. As h' is not effectively acyclic lemma 1 provides us a path ns to n where $ns \subseteq h'$, $n \in ns$ and $i \xrightarrow{ns}_{e'} n \neq 0$.

If $ns \subseteq h_k'$ for some $k \in \{1, 2\}$ then we already found our contradiction as h_k' is effectively acyclic by assumption: we can decompose ns into $ns = ns_1 \cdot n \cdot ns_2$ due to $n \in ns$. Then we obtain the contradiction

$$\begin{aligned}
&0 < i \xrightarrow{ns}_{e'} n \\
&= i' \xrightarrow{ns}_{e'} n && \text{by } i = i' \\
&\leq i_k' \xrightarrow{ns}_{e'} n && \text{by } \texttt{inf_fg_le_inf_fg} \text{ and monotonicity} \\
&\leq f_k' \xrightarrow{ns}_{e'} n && \text{by } \texttt{inf_fg_le_flow_fg} \text{ and monotonicity} \\
&= f_k' \xrightarrow{ns_1}_{e'} \xrightarrow{n \cdot ns_2}_{e'} n && \text{by decomposition of } ns \\
&\leq f_k' \xrightarrow{n \cdot ns_2}_{e'} n && \text{by } \texttt{chain_flow_fg_le_flow_fg} \\
&= 0 && \text{by effective acyclicity of } h_k'
\end{aligned}$$

Therefore, for the remaining proof we can assume that ns is nonempty and that $ns \cap h_k' \neq \emptyset$ for both $k = 1, 2$. From this we obtain a convoluted list nss using `split_segments` of at least two non-empty path segments that alternatingly belong to either h_1' or h_2', and $\mathsf{concat}\, nss = ns$. Without loss of generality, we assume that the first path segment of nss belongs to h_1'. If the first path segment of nss belongs to h_2' then the remaining proof of this case is symmetric with flipped flow graph indices.

From $\mathsf{concat}\, nss = ns$ we obtain the equality

$$i_1' \xrightarrow{ns}_{e'} n \ = \ i_1' \xrightarrow{nss}_{e'} n$$

Using this equality and `inf_fg_le_inf_fg` it follows

$$0 < i \xrightarrow{ns}_{e'} n = i' \xrightarrow{ns}_{e'} n \leq i'_1 \xrightarrow{ns}_{e'} n = i'_1 \xrightarrow{nss}_{e'} n$$

From this we can infer using `eff_acyclic_chain_length_le_card'` that each list segment of nss has alternatingly length of at most $|h_1|$ and $|h_2|$, respectively.

As we know that nss contains at least two path segments we now can consider the following two cases separately: $n \in \mathsf{last}\, nss$ and $n \notin \mathsf{last}\, nss$. Additionally, we know that $\mathsf{butlast}\, nss$ is nonempty and that each path segment in $\mathsf{butlast}\, nss$ is non-empty. Furthermore, we know that $\mathsf{last}\, nss$ is non-empty.

Case 1, EA, $n \in \mathsf{last}\, nss$ The case distinction provides us with the fact $n \in \mathsf{last}\, nss$. Additionally, we can assume that without loss of generality that $\mathsf{hd}\,(\mathsf{last}\, nss) \in h_1$. Otherwise, the proof simply flips h_1 and h_2 in the appropriate places.

From $\mathsf{hd}\,(\mathsf{last}\, nss) \in h_1$ we can infer that $\mathsf{last}\, nss \subseteq \mathsf{dom}\, h_1 = \mathsf{dom}\, h'_1$. Then:

$$\begin{aligned}
&0 < i'_1 \xrightarrow{nss}_{e'} n \\
&= i'_1 \xrightarrow{\mathsf{butlast}\, nss}_{e'} \xrightarrow{\mathsf{last}\, nss}_{e'} n \\
&\leq i'_1 \xrightarrow{\mathsf{last}\, nss}_{e'} n \\
&= i'_1 \xrightarrow{\mathsf{last}\, nss}_{e'_1} n && \text{by } \mathsf{last}\, nss \subseteq h'_1 \text{ and } h' = h'_1 + h'_2 \\
&\leq f'_1 \xrightarrow{\mathsf{last}\, nss}_{e'_1} n && \text{by } \texttt{inf_fg_le_flow_fg} \\
&\dots
\end{aligned}$$

We can decompose $\mathsf{last}\, nss$ into $\mathsf{last}\, nss = ns_1 \cdot n \cdot ns_2$ because $n \in \mathsf{last}\, nss$. Then

$$\begin{aligned}
&\dots \\
&= f'_1 \xrightarrow{ns_1}_{e'_1} \xrightarrow{n \cdot ns_2}_{e'_1} n && \text{by decomposition} \\
&\leq f'_1 \xrightarrow{n \cdot ns_2}_{e'_1} n && \text{by endomorphism} \\
&= 0 && \text{by effective acyclicity of } h'_1
\end{aligned}$$

This provides us the required contradiction and we are done with this case.

Note that compared to [Krishna et al., 2020] our proof avoids the transfer of parts of the convoluted path nss from h' to h by exploiting the observation that $\mathsf{int}\, h = \mathsf{int}\, h'$. This observation enables us to directly approximate $i'_1 \xrightarrow{\mathsf{butlast}\, nss}_{e'}$ with i'_1 as inflow and path refer to the same flow graph $h' = h'_1 + h'_2$. In contrast, [Krishna et al., 2020] approximates the flow along nss with a chain of capacities in order to be able to execute the transfer from h' and h which in turn enables an approximation comparable to ours within h.

In case $n \notin \mathsf{last}\, nss$ we can not apply the same idea because there the cycle is not constrained to a single subgraph but involves multiple path segments through both subgraphs, i.e. we have to conduct our contradiction against the effective acyclicity of h instead of being able to use the effective acyclicity of h'_1 for this.

Case 1, EA, $n \notin \mathsf{last}\, nss$

From $n \in ns$ we know that there must be some decomposition $nss = nss_1 \cdot (ns_1 \cdot n \cdot ns_2) \cdot nss_2$. nss_2 is non-empty because $n \notin \mathsf{last}\, nss$. Additionally, $\mathsf{hd}\, nss_2$ is non-empty because all segments in nss are non-empty.

Using the decomposition we obtain

$$\begin{aligned} 0 < i &\xrightarrow{ns}_{e'} n \\ &= i \xrightarrow{nss_1 \cdot (ns_1 \cdot n \cdot ns_2) \cdot nss_2}_{e'} n \\ &= (i \xrightarrow{nss_1 \cdot ns_1}_{e'} n) \xrightarrow{n \cdot ns_2 \cdot nss_2}_{e'} n \end{aligned}$$

Using pointwise reduction we then obtain

$$\begin{aligned} 0 \neq ((i \xrightarrow{nss_1 \cdot ns_1}_{e'} n) \xrightarrow{n \cdot ns_2 \cdot nss_2}_{e'} n) \xrightarrow{n \cdot ns_2 \cdot nss_2}_{e'} n \\ = ((i \xrightarrow{nss_1 \cdot ns_1}_{e'} n) \xrightarrow{n \cdot ns_2 \cdot nss_2}_{e'} n) \xrightarrow{n \cdot ns_2}_{e'} \xrightarrow{nss_2}_{e'} n \end{aligned}$$

In this inequality we can omit the last arrow because arrows are endomorphisms and for endomorphisms holds $e_1\ (e_2\ x) \neq 0 \Longrightarrow e_2\ x \neq 0$. Then:

$$\begin{aligned} 0 \neq ((i \xrightarrow{nss_1 \cdot ns_1}_{e'} n) \xrightarrow{n \cdot ns_2 \cdot nss_2}_{e'} n) \xrightarrow{n \cdot ns_2}_{e'} \mathsf{hd}\,(\mathsf{hd}\, nss_2) \\ = (i \xrightarrow{nss_1 \cdot ns_1}_{e'} n) \xrightarrow{n \cdot ns_2 \cdot nss_2 \cdot n \cdot ns_2}_{e'} \mathsf{hd}\,(\mathsf{hd}\, nss_2) \\ = i \xrightarrow{nss_1 \cdot ns_1 \cdot n \cdot ns_2 \cdot nss_2 \cdot n \cdot ns_2}_{e'} \mathsf{hd}\,(\mathsf{hd}\, nss_2) \end{aligned}$$

We know that $\mathsf{concat}\,(nss_1 \cdot ns_1 \cdot n \cdot ns_2 \cdot nss_2) = \mathsf{concat}\, nss = ns$ is an alternating list. We have to extend this alternating list using `split_segments_extend` to an alternating list mss with $ns \cdot n \cdot ns_2 = \mathsf{concat}\,(nss_1 \cdot (ns_1 \cdot n \cdot ns_2) \cdot nss_2 \cdot (n \cdot ns_2))$. As $\mathsf{last}\, nss_2$ and $n \cdot nss_2$ can be in the same subgraph or in different subgraphs we do not know if $n \cdot ns_2$ is merged with $\mathsf{last}\, nss_2$ or becomes an additional path segment. To account for these two possibilities we introduce two auxiliary variables zs and zss where exactly one of those two variables equals [] and the other one equals $n \cdot ns_2$ and $[n \cdot ns_2]$, respectively. Then $mss = nss_1 \cdot (ns_1 \cdot n \cdot ns_2) \cdot \mathsf{butlast}\, nss_2 \cdot (\mathsf{last}\, nss_2 \cdot zs) \cdot zss$. For the next derivation we introduce some abbreviations:

$$\lambda_1\ xs\ y \coloneqq \begin{cases} \mathsf{cap}'|h_1|\ h_1\ e'\ (\mathsf{hd}\,(xs \cdot y))\ y & \text{for } xs \subseteq h_1 \\ \mathsf{cap}'|h_1|\ h_2\ e'\ (\mathsf{hd}\,(xs \cdot y))\ y & \text{for } xs \subseteq h_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\lambda_2\ xs\ y \coloneqq \begin{cases} \mathsf{cap}'|h_1|\ h_1\ e_1'\ (\mathsf{hd}\,(xs \cdot y))\ y & \text{for } xs \subseteq h_1 \\ \mathsf{cap}'|h_1|\ h_2\ e_2'\ (\mathsf{hd}\,(xs \cdot y))\ y & \text{for } xs \subseteq h_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\lambda_3\ xs\ y \coloneqq \begin{cases} \mathsf{cap}'|h_1|\ h_1\ e_1\ (\mathsf{hd}\,(xs \cdot y))\ y & \text{for } xs \subseteq h_1 \\ \mathsf{cap}'|h_1|\ h_2\ e_2\ (\mathsf{hd}\,(xs \cdot y))\ y & \text{for } xs \subseteq h_2 \\ 0 & \text{otherwise} \end{cases}$$

Additionally, we extend the arrow notation: for $xss = [xs_1, \ldots, xs_n]$

$$m \xrightarrow{xss}\!\!\!\!\!\rightarrow_{\lambda_k} y \equiv m \triangleright \lambda_k\ xs_1\ (\mathsf{hd}\, xs_2) \triangleright \cdots \triangleright \lambda_k\ xs_n\ y$$

Then:

$$0 < i \xrightarrow{nss_1 \cdot (ns_1 \cdot n \cdot ns_2) \cdot nss_2 \cdot (n \cdot ns_2)}_{e'} \mathsf{hd}\,(\mathsf{hd}\, nss_2)$$

$$
\begin{aligned}
&= i \xrightarrow{mss}_{e'} \mathsf{hd}\,(\mathsf{hd}\,nss_2) \\
&\leq i \xrightarrow{mss}\!\!\!\!\!\rightarrow_{\lambda_1} \mathsf{hd}\,(\mathsf{hd}\,nss_2) \\
&\leq i \xrightarrow{mss}\!\!\!\!\!\rightarrow_{\lambda_2} \mathsf{hd}\,(\mathsf{hd}\,nss_2) \\
&\leq i \xrightarrow{mss}\!\!\!\!\!\rightarrow_{\lambda_3} \mathsf{hd}\,(\mathsf{hd}\,nss_2) \\
&\leq i_1 \xrightarrow{mss}\!\!\!\!\!\rightarrow_{\lambda_3} \mathsf{hd}\,(\mathsf{hd}\,nss_2) \\
&= i_1 \xrightarrow{nss_1 \cdot ns_1 \cdot n \cdot ns_2}\!\!\!\!\!\rightarrow_{\lambda_3} \xrightarrow{\mathsf{butlast}\,nss2 \cdot (\mathsf{last}\,nss2 \cdot zs) \cdot zss}\!\!\!\!\!\rightarrow_{\lambda_3} \mathsf{hd}\,nss_2 \\
&\leq i(\mathsf{alt}\,h_2\ h_1\ (nss_1 \cdot ns_1 \cdot n \cdot ns_2)) \xrightarrow{\mathsf{butlast}\,nss2 \cdot (\mathsf{last}\,nss2 \cdot zs) \cdot zss}\!\!\!\!\!\rightarrow_{\lambda_3} \mathsf{hd}\,nss_2 \\
&\leq \sum_{xs} i_1 \xrightarrow{\mathsf{hd}\,(\mathsf{hd}\,nss2) \cdot xs}_{e} \mathsf{hd}\,(\mathsf{hd}\,nss_2)
\end{aligned}
$$

The first inequality is known from above. The next one simply rewrites the arrow using the decomposition of mss. The third one approximates each path segment with a capacity using `chain_le_cap'` and monotonicity. The fourth one substitutes e'_1 or e'_2 for e' as each capacity is in either h'_1 or h'_2. The fifth one transfers each capacity from h'_k to h_k by applying lemma 3. The sixth one applies `inf_fg_le_inf_fg`. Then we split the arrow using `chains'_append`. Then we apply `chains'_cap_fg_inf_fg_le_inf_fg'`. Then we apply `chains'_cap_inf_le_chain_sum`.

This inequality implies that there is some path xs such that

$$i_1 \xrightarrow{\mathsf{hd}\,(\mathsf{hd}\,nss2) \cdot xs}_{e} \mathsf{hd}\,(\mathsf{hd}\,nss_2) \neq 0$$

This proposition contradicts effective acyclicity of h.

Case 1, $\preccurlyeq_S$

First we notice that $\mathsf{int}\,h \preccurlyeq \mathsf{int}\,h'$ holds trivially due to $\mathsf{int}\,h = \mathsf{int}\,h'$. It remains to show that

$$\forall n \in h, n' \in -h', m \leq \mathsf{inf}\,h\ n.\ m \triangleright \mathsf{cap}\,h\ n\ n' = m \triangleright \mathsf{cap}\,h'\ n\ n'$$

This is exactly what lemma 6 states.

Case 2: $\mathsf{dom}\,h_1 \subsetneq \mathsf{dom}\,h_2$

We now consider the case where $\mathsf{dom}\,h_1 \subsetneq \mathsf{dom}\,h_2$. The idea is to reduce this case to case $\mathsf{dom}\,h_1 = \mathsf{dom}\,h_2$. Therefore, we have to construct a flow graph h''_1 such that we can apply case $\mathsf{dom}\,h_1 = \mathsf{dom}\,h_2$ for $h_1 \coloneqq h''_1$ and $h_2 = h_2$.

We define $h_0 \coloneqq (N_0, e_0, f_0) \coloneqq (\mathsf{dom}\,h'_1 \setminus \mathsf{dom}\,h_1, \lambda n\,n'\ \_.\ 0, \lambda n.\ i'_1\ n + o_1\ n)$. Using this we define $h''_1 \coloneqq (N''_1, e''_1, f''_1) \coloneqq h_0 + h_1$. We prove that $h''_1 \neq \bot$ using `fgI`, i.e. we have to show that $\mathsf{dom}\,h''_1$ is finite and that the flow equation holds for h''_1 and i'_1: $\mathsf{dom}\,h''_1$ is finite because the domain of h''_1 is composed of finite domains of flow graphs. The flow equation holds for $n \in h_0$ because

$$
\begin{aligned}
f''_1\ n &= f_0\ n && \text{by def. } f''_1 \text{ on } N_0 \\
&= i'_1\ n + o_1\ n + 0 && \text{by def. of } f_0 \\
&= i'_1\ n + (\sum_{n' \in N_1} f_1\ n' \triangleright e_1\ n'\ n) + (\sum_{n' \in N_0} f''_1\ n' \triangleright e''_1\ n'\ n) && \text{by def. of } o_1 \text{ and } e''_1 \\
&= i'_1\ n + (\sum_{n' \in N_1} f''_1\ n' \triangleright e''_1\ n'\ n) + (\sum_{n' \in N_0} f''_1\ n' \triangleright e''_1\ n'\ n) && \text{by def. of } f''_1 \text{ on } N_1
\end{aligned}
$$

$$= i'_1\ n + (\sum_{n' \in N_0 \cup N_1} f''_1\ n' \triangleright e''_1\ n'\ n) \qquad \text{by disj. of } N_0 \text{ and } N_1$$

and for $n \in h_1$ because

$$\begin{aligned}
f''_1\ n &= f_1\ n && \text{by def. of } f''_1 \text{ on } N_1 \\
&= i_1\ n + (\sum_{n' \in h_1} f_1\ n' \triangleright e_1\ n'\ n) + 0 && \text{by def. of flow eq.} \\
&= i'_1\ n + (\sum_{n' \in h_1} f_1\ n' \triangleright e_1\ n'\ n) + (\sum_{n' \in h_0} f''_1\ n' \triangleright e''_1\ n'\ n) && \text{by } i_1 = i'_1 \text{ and def. of } e''_1 \\
&= i'_1\ n + (\sum_{n' \in h_1} f''_1\ n' \triangleright e''_1\ n'\ n) + (\sum_{n' \in h_0} f''_1\ n' \triangleright e''_1\ n'\ n) && \text{by def. of } f''_1 \text{ on } N_1 \\
&= i'_1\ n + (\sum_{n' \in N_0 \cup N_1} f''_1\ n' \triangleright e''_1\ n'\ n) && \text{by disj. of } N_0 \text{ and } N_1
\end{aligned}$$

Therefore, we have $h''_1 = h_0 + h_1 \neq \bot$. Note that $\mathsf{inf}\, h''_1 = i'_1$. Using lemma 7 we obtain that h''_1 is effectively acyclic.

Next, we show that $h''_1 \preccurlyeq_S h'_1$:

- $\mathsf{int}\, h''_1 = \mathsf{int}\, h'_1$: we have $\mathsf{dom}\, h''_1 = \mathsf{dom}\, h'_1$, therefore $\mathsf{dom}\, h''_1 \subseteq \mathsf{dom}\, h'_1$. For all $n \in \mathsf{dom}\, h''_1 = \mathsf{dom}\, h'_1$ it holds that $\mathsf{inf}\, h''_1\ n = i'_1\ n = \mathsf{inf}\, h'_1\ n$. By lemma 8 h''_1 and h'_1 have the same outflows.
- $\forall n \in h''_1,\ n' \in -h'_1.\ m \leq \mathsf{inf}\, h''_1\ n \longrightarrow m \triangleright \mathsf{cap}\, h''_1\ n\ n' = m \triangleright \mathsf{cap}\, h'_1\ n\ n'$: for $n \in h_0$ we have

  $$\begin{aligned}
  m \triangleright \mathsf{cap}\, h''_1\ n\ n' &= 0 && \text{by lemma 9} \\
  &= m \triangleright \mathsf{cap}\, h'_1\ n\ n'' && \text{by } h_1 \preccurlyeq h'_1 \text{ and } n \in \mathsf{dom}\, h'_1 - \mathsf{dom}\, h_1
  \end{aligned}$$

  and for $n \in h_1$ we have

  $$\begin{aligned}
  m \triangleright \mathsf{cap}\, h''_1\ n\ n' &= m \triangleright \mathsf{cap}\, h''_1\ n\ n' && \text{by lemma 9} \\
  &= m \triangleright \mathsf{cap}\, h'_1\ n\ n'' && \text{by } h_1 \preccurlyeq h'_1 \text{ and } n \in \mathsf{dom}\, h_1
  \end{aligned}$$

- $\forall n \in \mathsf{dom}\, h'_1 - \mathsf{dom}\, h''_1. \ldots.$: holds trivially true due to $\mathsf{dom}\, h''_1 = \mathsf{dom}\, h'_1$.

Now we have to repeat the last page to show that $h'' = h''_1 + h_2 = h_0 + h_1 + h_2 = h_0 + h$ exists and is effectively acyclic. We prove $h'' \neq \bot$ using `fgI`: $\mathsf{dom}\, h''_1 \cup \mathsf{dom}\, h_2$ is finite because it is constructed using finite domains. Now we prove that the flow equation holds for h'' and i'' where $i'' \coloneqq \lambda n.\ \texttt{if}\ n \in N\ \texttt{then}\ i\ n\ \texttt{else}\ i'_1\ n$. This time we have to consider three cases: first, $n \in h_1$:

$$\begin{aligned}
f''\ n &= f_1\ n = f\ n && \text{by def. of sums} \\
&= i\ n + (\sum_{n' \in h} f\ n' \triangleright e\ n'\ n) && \text{by flow eq. for } h \\
&= i\ n + (\sum_{n' \in h_1} f\ n' \triangleright e\ n'\ n) + (\sum_{n' \in h_2} f\ n' \triangleright e\ n'\ n) && \text{by } h = h_1 + h_2 \\
&= i\ n + (\sum_{n' \in h_1} f''\ n' \triangleright e''\ n'\ n) + (\sum_{n' \in h_2} f''\ n' \triangleright e''\ n'\ n) && \text{by def. of } e'', f'' \text{ on } h_1, h_2
\end{aligned}$$

$$
\begin{aligned}
&= i\ n + (\sum_{n' \in h_1 \cup h_2} f''\ n' \triangleright e''\ n'\ n) && \text{by } h = h_1 + h_2 \\
&= i\ n + (\sum_{n' \in h_1 \cup h_2} f''\ n' \triangleright e''\ n'\ n) + (\sum_{n' \in h_0} f''\ n' \triangleright e''\ n'\ n) && \text{by def. of } e'' \text{ on } h_0 \\
&= i\ n + (\sum_{n' \in h_0 \cup h_1 \cup h_2} f''\ n' \triangleright e''\ n'\ n) && \text{by } h_1' \cap h_2 = \emptyset, h_1'' = h_0 + h_1, \\
&= i''\ n + (\sum_{n' \in h''} f''\ n' \triangleright e''\ n'\ n) && \text{by def. of } i'' \text{ for } n \in h_1
\end{aligned}
$$

Second, $n \in h_2$ follows analogously except for the first two steps: $f''\ n = f_2\ n = f\ n$. Third, $n \in h_0$:

$$
\begin{aligned}
f''\ n &= f_0\ n && \text{by def. of } f'', n \in h_0 \\
&= i_1'\ n + o_1\ n && \text{by def. of } f_0 \\
&= i_1'\ n + (\sum_{n' \in h_1} f_1\ n' \triangleright e_1\ n'\ n) && \text{by def. of outflow} \\
&= i_1'\ n + (\sum_{n' \in h_1} f''\ n' \triangleright e''\ n'\ n) && \text{by } h_1'' = h_0 + h_1,\ n' \in h_1 \\
&= i_1'\ n + (\sum_{n' \in h_1} f''\ n' \triangleright e''\ n'\ n) + (\sum_{n' \in h_0} f''\ n' \triangleright e''\ n'\ n) && \text{by } e''\ n'\ \_ = 0 \text{ for } n' \in h_0 \\
&= i_1'\ n + (\sum_{n' \in h_0 \cup h_1} f''\ n' \triangleright e''\ n'\ n) && \text{by } h_1'' = h_0 + h_1 \\
&= i_1'\ n + (\sum_{n' \in h_0 \cup h_1} f''\ n' \triangleright e''\ n'\ n) + (\sum_{n' \in h_2} f''\ n' \triangleright e''\ n'\ n) && \text{by } e''\ n'\ \_ = 0 \text{ for } n \in h_2 \\
&= i_1'\ n + (\sum_{n' \in h_0 \cup h_1 \cup h_2} f''\ n' \triangleright e''\ n'\ n) && \text{by domains disjoint} \\
&= i''\ n + (\sum_{n' \in h''} f''\ n' \triangleright e''\ n'\ n) && \text{by def. of } i''
\end{aligned}
$$

Therefore, $h'' \neq \bot$. lemma 7 again provides us that h'' is effectively acyclic.

Now, we can invoke case $\mathsf{dom}\, h_1' = \mathsf{dom}\, h_2$ and obtain some effectively acyclic flow graph $h' = h_1' + h_2 \neq \bot$ such that $h'' \preccurlyeq_S h'$. It remains to show that $h \preccurlyeq_S h'$. By transitivity of subflow-preserving extensions it is sufficient to prove that $h \preccurlyeq_S h''$ as we already know that $h'' \preccurlyeq_S h'$:

- $\mathsf{int}\, h \preccurlyeq \mathsf{int}\, h''$: $\mathsf{dom}\, h \subseteq \mathsf{dom}\, h''$ follows from the definition of h''. $\forall n \in \mathsf{int}\, h.\ i\ n = i''\ n$ follows from the definition of i'' for $n \in h$. $\forall n \in -\mathsf{int}\, h''.\ o\ n = o''\ n$ follows from lemma 8.
- $\forall n \in h, n' \in -h'', m \leq i\ n.\ \mathsf{cap}\, h\ n\ n'\ m = \mathsf{cap}\, h''\ n\ n'\ m$: This follows immediately fom lemma 9.
- $\forall n \in h'' - h, n' \in -h'', m \leq i''\ n.\ \mathsf{cap}\, h''\ n\ n'\ m = 0$: This follows immediately fom lemma 9.

■

7. Conclusion

7.1. Summary

This thesis verified all important results on the foundational and extended Flow Framework from [Krishna et al., 2020] and applied the Framework to verify the so-called PIP example from [Krishna et al., 2020].

Our formalization of the Flow Framework in Isabelle/HOL confirms the soundness of the theoretical foundation of the Flow Framework. We identified and fixed several small formal flaws in the existing paper proofs in [Krishna, 2019]. The proof of [Krishna et al., 2020, th. 3], which unexpectedly turned out to become the primary contribution of this thesis, was proven invalid by a counter-example and fixed by introducing a stronger notion of reduced endomorphisms. The proof required us to come up with multiple non-trivial steps complementing the original high-level paper proof in [Krishna, 2019, th 3.38]. We formalized and introduced multiple levels of abstraction in order to obtain building blocks that reasonably match the high abstraction level of the original paper proof. We provided the first formalization of the advanced theories (in particular effective acyclicity) about the Flow Framework. To support future verification efforts involving this proof we provided a significantly refined paper proof that hopefully captures all necessary details to enhance or reproduce our proof.

We verified the PIP example using the Flow Framework and extended the proof by additionally showing termination. Our termination proof exploits information provided by the flow graph and and demonstrates that the Flow Framework is able to prove global properties of data structures without reverting to global reasoning. We furthermore demonstrated how to use the Flow Framework to obtain abstract information that can be used in global reasoning style proofs without encountering spatial issues caused by separation logic. We identified multiple difficulties regarding the usability of our implementation of the Flow Framework that should be addressed by future development.

7.2. Future Work

- The formalization of [Krishna et al., 2020, theorem 3] was far more challenging than initially expected. The proof has to be considered a proof-of-concept. Especially regarding reasoning on alternating paths there are many proof steps that should be cleaned up and simplified or generalized into more powerful primitives on alternating lists. Also, reasoning about overlaid list representations, e.g. $xs = xs_1 \cdot xs_2 = ys_1 \cdot ys_2 \cdot ys_3$, and expanding those introduced quite some disturbance in our proof. There might be more concise approaches than ours.

- In our PIP example we were significantly challenged by reasoning on flow interfaces. Especially larger sums of flow interfaces required significant manual intervention, in particular in interaction with the fake interface proof technique. In order to simplify or avoid this reasoning an effective suite of simplification rules or proof methods would be highly useful.

- In this context we are not sure if our approach of opaquely defining flow graphs and interfaces might not compromise automated reasoning about these structures. Similarly, opaque definitions of node and graph abstractions for heap locations might affect automatic reasoning. A comparison to other applications of the Flow Framework like [Krishna and Wies, 2020] might provide some insights for more efficient definitions and proof methods. A glimpse into their verification of algorithms using Coq and Iris suggests that they have to apply similar rigor as we do in our PIP example verification. We think another interesting question is how the effort of verification and flow interface reasoning embedded into theorem provers compares to more specialized verifiers like GRASShopper [gra, 2020] which is also used by [Krishna and Wies, 2020]. Also, we asked ourselves the question if in our case automation and simplification maybe caused more disturbance than benefit regarding flow interface computations compared to a more rigor manual application of proof rules.

- Initially, we planned to apply the Flow Framework to conduct a comparison with an existing verification of fibonacci trees using Separation Logic. Due to the unexpected challenges this plan was not realizable in to given time frame. This project would be an interesting sequel to this thesis that also could concentrate on and generate progress towards more and better automation.

- During the course of this thesis we came up with the question on local invariants and their relationship to global invariants. We already stated some thoughts about this topic in Section 5.2.10. Maybe in this spot there are some additional interesting insights hidden.

- Regarding our counter-example for theorem 3 of [Krishna et al., 2020] it is an open question whether there is a weaker variant of reduction than point-wise reduction that enables the proof.

A. Project Structure

In this chapter we shortly describe the structure of our project and its relationship to this thesis.

As the central deliverable of our thesis we provide an Isabelle session named `flow`. The session was developed using Isabelle2020 [isa, 2020] and the AFP version 2020-08-15 [afp, 2020]. This session contains our development on the Flow Framework:

- Theory `Auxiliary` contains some basic notations. Furthermore, it contains a lot of random results on lists and sets that are not of particular interest.
- Theories `Alternating`, `Chain`, `Endomorphism` and `Pigeonhole` contain the preliminary development sketched in Chapter 3.
- Theories `Flow_Graph` and `Flow_Interface` contain the formalizations of flow graphs (subsections 4.1.2 and 4.1.3) and interfaces (subsections 4.1.5 and 4.1.6). Theory `Flow` then formalizes their connection (subsection 4.1.7). Theories `Flow_Graph_Separation_Algebra` and `Flow_Interface_Separation_Algebra` contain the proofs that our formalizations of flow graphs and interfaces are separation algebras if constrained to valid values. Theory `Flow_Extensions` formalizes contextual and subflow-preserving extensions and proves the replacement theorem.
- Theories `Edge_Local` and `Nilpotent` contain the formalizations of subsections 4.2.1 and 4.2.2.
- The definition of effective acyclicity and related auxiliary lemmas are formalized in `Effectively_Acyclic`. The uniqueness of effectively acyclic flow graphs and the counter-example presented in subsection 4.2.3 are formalized in theories `Effectively_Acyclic`, `Effectively_Acyclic_Uniqueness` and `Effectively_Acyclic_Maintain_Counter_Example`. Theories `Effectively_Acyclic_Approximations`, `Effectively_Acyclic_Equal_Capacity` and `Effectively_Acyclic_Switch_Worlds` and `Effectively_Acyclic_Source_Path` contain the formalizations of sections 6.2 to 6.5. Finally, theory `Effectively_Acyclic_Maintain` contains our proof of the existence of flow for effectively acyclic flow graphs from sections 6.6 and 6.7.
- Theory `FF` contains our formalization of our integration of the Flow Framework into the Separation Logic framework (section 5.1).
- Theory `PIP_Shared` contains the instantiation of the Flow Framework and the definitions of the operations of our example (subsections 5.2.2 and 5.2.3). Theories `PIP_Release` and `PIP_Acquire` prove correctness of the release, acquire and update operations of the PIP example (subsection 5.2.8). Theory `PIP_Example` proves the small example from subsection 5.2.5. Theory `PIP_Invariant` formalizes subsection 5.2.10.

Additionally, in folder `aux` there formalizations of our examples on Hoare logic and Separation Logic from chapter 2. Folder `thesis` contains the LaTeX code of this thesis.

Bibliography

[gra, 2020] (2020). Archive of formal proofs. `https://cs.nyu.edu/wies/software/grasshopper`, accessed 2020-09-14.

[afp, 2020] (2020). Archive of formal proofs. `https://www.isa-afp.org/`, accessed 2020-09-13.

[isa, 2020] (2020). Isabelle. `http://isabelle.in.tum.de/`, accessed 2020-09-13.

[Antonopoulos et al., 2014] Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M. I., and Ouaknine, J. (2014). Foundations for decision problems in separation logic with general inductive predicates. In Muscholl, A., editor, *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 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 8412 of *Lecture Notes in Computer Science*, pages 411–425. Springer.

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

[Calcagno et al., 2007] Calcagno, C., O'Hearn, P. W., and Yang, H. (2007). Local action and abstract separation logic. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 366–378, Washington, DC, USA. IEEE Computer Society.

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

[Hobor and Villard, 2013] 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.

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

[Katelaan et al., 2019] 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.

[Klein et al., 2012] Klein, G., Kolanski, R., and Boyton, A. (2012). Separation algebra. *Archive of Formal Proofs*. `http://isa-afp.org/entries/Separation_Algebra.html`, Formal proof development.

[Krishna, 2019] Krishna, S. (2019). Compositional abstractions for verifying concurrent data structures.

[Krishna et al., 2019] Krishna, S., Summers, A. J., and Wies, T. (2019). Local reasoning for global graph properties. *ArXiv*, abs/1911.08632.

[Krishna et al., 2020] Krishna, S., Summers, A. J., and Wies, T. (2020). Local reasoning for global graph properties. In Müller, P., editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 308–335. Springer.

[Krishna and Wies, 2020] Krishna, S. and Wies, T. (2020). Artifact of verifying concurrent search structure templates. `https://github.com/nyu-acsys/template-proofs`, accessed 2020-09-11.

[Lammich and Meis, 2012] Lammich, P. and Meis, R. (2012). A separation logic framework for imperative hol. *Archive of Formal Proofs*. `http://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html`, Formal proof development.

[Nipkow and Klein, 2014] Nipkow, T. and Klein, G. (2014). *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated.

[O'Hearn, 2019] O'Hearn, P. (2019). Separation logic. *Commun. ACM*, 62(2):86–95.

[O'Hearn, 2007] O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307.

[O'Hearn and Pym, 1999] O'Hearn, P. W. and Pym, D. J. (1999). The logic of bunched implications. *Bull. Symbolic Logic*, 5(2):215–244.

[Pym et al., 2018] Pym, D., Spring, J., and O'Hearn, P. (2018). Why separation logic works. *Philosophy and Technology*.

[Reynolds, 2002] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA. IEEE Computer Society.

[Sha et al., 1990] Sha, L., Rajkumar, R., and Lehoczky, J. P. (1990). Priority inheritance protocols: an approach to real-time synchronization.

[Steger, 2007] Steger, A. (2007). *Diskrete Strukturen: Band 1: Kombinatorik, Graphentheorie, Algebra*. Springer-Lehrbuch. Springer Berlin Heidelberg.

[Vafeiadis, 2011] Vafeiadis, V. (2011). Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351.

[Vafeiadis and Parkinson, 2007] 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.