Imperial College London
Department of Computing

Structural Separation Logic

Adam Douglas Wright

August 2013

Supervised by Professor Philippa Gardner

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Adam Douglas Wright

Abstract

This thesis presents structural separation logic, a novel program reasoning approach for software that manipulates both standard heaps and structured data such as lists and trees. Structural separation logic builds upon existing work in both separation logic and context logic. It considers data abstractly, much as it is exposed by library interfaces, ignoring implementation details.

We provide a programming language that works over structural heaps, which are similar to standard heaps but allow data to be stored in an abstract form. We introduce abstract heaps, which extend structural heaps to enable local reasoning about abstract data. Such data can be split up with structural addresses. Structural addresses allow sub-data (e.g. a sub-tree within a tree) to be abstractly allocated, promoting the sub-data to an abstract heap cell. This cell can be analysed in isolation, then re-joined with the original data. We show how the tight footprints this allows can be refined further with *promises*, which enable abstract heap cells to retain information about the context from which they were allocated. We prove that our approach is sound with respect to a standard Hoare logic.

We study two large examples. Firstly, we present an axiomatic semantics for the Document Object Model in structural separation logic. We demonstrate how structural separation logic allows abstract reasoning about the DOM tree using tighter footprints than were possible in previous work. Secondly, we give a novel presentation of the POSIX file system library. We identify a subset of the large POSIX standard that focuses on the file system, including commands that manipulate both the file heap and the directory structure. Axioms for this system are given using structural separation logic. As file system resources are typically identified by paths, we use promises to give tight footprints to commands, so that that they do not require all the resource needed to explain paths being used. We demonstrate our reasoning using a software installer example.

For those who knew that I could when I thought that I couldn't.

Contents

List of Figures

1. Introduction and background

> *"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise"*
>
> Edsger Dijkstra, *The Humble Programmer* [20]

In this thesis, we link local reasoning about heaps to local reasoning about rich data structures, and so create *structural separation logic*. Our aim is *library reasoning*: the ability to give clear and useful axiomatic specifications to programming libraries concerned with manipulating structured data. We focus on *data abstractions*, and so create axioms that capture the effect of commands in a natural manner. This leads to easier proof construction, and the ability to give specifications that match programmers' intuitions. Just as libraries enhance programming languages and ease the development of programs, structural separation logic enhances separation logic [52] and eases the verification of programs.

Programming libraries are abstractions for sets of operations. They are the standard method for interacting with both data structures and system services. It is rare to see a programmer using, for example, a custom list implementation. Instead, they use a pre-existing list library, which abstracts away from the implementation details. This allows programmers to use lists without worrying about how they are implemented. An operation such as removing element e from a list referenced by l is not exposed as *"If you pass in a list reference l and element e, this command obtains an initial linked-list pointer from l, walks along a doubly-linked structure until the memory containing element e is encountered, swings the pointers around it, and deallocates that memory"*. Rather, it is exposed as *"If you pass in list reference l and element e, the list at l will be updated such that element e has been removed, and nothing else has changed"*. The first description is much too "low level". It focuses on how an operation is achieved rather than what the operation does, and it forces the programmer to think in terms of machine memory. The second is "high level" (equally, at the "level of the abstraction"). It allows a programmer to focus on the *meaning* of the code they are writing.

Many libraries are exposed at a high level. From the programmer's perspective, these libraries do not set out to abstract some fixed implementation, but instead implement a well-chosen abstraction. How the operations are performed is hidden, allowing the library to change implementation details without changing its interface. Moreover, and perhaps more importantly, this allows programmers to use the library with an mental model of what it does, rather than how it does it.

Often, libraries are defined only in these abstract terms and have no standard implementation. If you obtained this thesis from the Internet, you probably downloaded it from a web page that contained JavaScript [27]. Web browsers expose their internal web page representation to JavaScript via the *Document Object Model* [71], allowing script programs embedded within the page to manipulate the in-memory representation. The DOM, a standardised programming library based upon trees, is described only via an abstract specification. Browser authors must carefully read the specification, and provide an implementation that matches the operations described. However, script programmers can simply rely on the abstract description of the trees and operations upon them. Another ubiquitous system presented abstractly is the *Portable Operating Systems Interfaces for UNIX* standard, commonly abbreviated to POSIX [1]. POSIX is a library that all UNIX-like operating systems offer. By targeting the POSIX library rather than OS specific interfaces, programmers can create software that should compile and run on any operating system that provides POSIX. In this thesis, we will demonstrate how structural separation logic can provide natural axiomatic semantics for both DOM and a subset of POSIX focusing on file systems.

Libraries help programmers perform informal reasoning about their code. Leveraging good abstractions, programmers can quickly determine the expected effect of their work and track down bugs. The informal nature of this approach inevitably leads to problems, which *formal reasoning* can help eliminate. Ideally, formal reasoning should (as far as possible) follow the insights provided by informal reasoning. Unfortunately, this has rarely been the case, and formal reasoning remains out of reach for most programmers. It is generally adopted only by cost-critical systems such as hardware design and the finance industry, or safety-critical systems such as aircraft and national defense [72]. The majority of software is never subjected to mathematical rigour, relying instead on manual testing.

This is not terribly surprising. Formal methods have not generally kept pace with software development practices. One very common programming technique is the use of *heap memory*. Formal reasoning about heap manipulating software has, traditionally, been badly served. Fortunately, over the last decade, this has been changing, with some heap-using programs being statically analysed before they are given to customers. For

example, the SLAM project [6] from Microsoft forms the basis of several tools used to analyse parts of their Windows operating system before they are shipped. Their Static Driver Verifier [5], which uses the SLAM engine, checks that device drivers cannot cause system failures. Going even further than this is the L4.verified project [49], which provides formal verification for an entire operating system kernel.

One theoretical development that is enabling the verification of heap manipulating programs *and* capturing the spirit of informal program justification is *separation logic* [52]. Separation logic is a Hoare logic [42] for *local reasoning*. In essence, local reasoning means that verifying a program requires consideration of only the parts of memory touched by that program. This *footprint* concept very often matches the intuitions a programmer has when writing heap manipulating code. When, for example, implementing a linked list reversal, the core loop deals only with the current list element, the next, and the previous. Programmers focus only on this memory, knowing that all other list elements will not change during the current loop iteration. Separation logic reasoning considers the same memory as the programmer, in contrast to previous reasoning techniques which often required consideration of the entire list (or even the entire heap). This similarity can give the creation of a separation logic proof a similar flavour to writing the code in the first place.

Separation logic achieves local reasoning by focusing the descriptions of program heaps onto individual machine memory cells. These *heap cells* can be combined with a *separating conjunction*. Each cell combined with the separating conjunction is guaranteed to be disjoint from all others, allowing the effects of heap update to be tracked precisely to the region of memory affected. Commands are then specified with *small axioms*, which describe the effect of the command over the smallest possible number of heap cells. This technique sidesteps many of the problems that traditional Hoare reasoning about heaps typically suffers, and allows commands to be specified using only the memory they will actually access - the command *footprint* alluded to earlier. The footprint concept has made separation logic highly compositional, allowing proofs to behave modularly. In the past decade, the local reasoning philosophy, and separation logic itself, has flourished. Many research sites now work with local reasoning, with uses spanning the verification of complex concurrent algorithms [19] to automated reasoning tools that can prove large amounts of real world code free of memory errors[1]. Separation logic has been particularly successful in proving properties of *concurrent* programs. Separation fits extremely well with concurrent data update, as manipulating disjoint data can never result in *data races*, where multiple program threads simultaneously access the same data.

[1]Such as over 50% of Apache, OpenSSH and Linux [24]

Separation logic's representation of data is typically built upon the shape of machine memory. When dealing with structured data, such as lists or binary trees, a variety of *data predicates* are used. The simplest approach is *inductive predicates*, which encode an implementation of the structure into the heap. Whilst inductive predicates allow inductively defined structures to be represented, they do not provide good *abstractions* of the data structure. The implementation details leak through the predicates, making proofs dependent on them. Approaches such as *abstract predicates* [57] hide more of the implementation choices, representing data structures as "black-boxes". These black-boxes can be manipulated via axioms, and only the library implementation can *unroll* the boxes to reveal implementation details.

We term these approaches "bottom-up". The abstractions must exist alongside machine heap assertions, will be justified in terms of the heap, and are manipulated similarly to heap cells. Unfortunately the notion of separation, being designed to support heap memory, can struggle when trying to enable local reasoning about these abstractions, especially when they are abstracting highly structured data. The natural specifications for commands over, e.g. lists or n-ary trees, often want to work on single list elements or sub-trees. Separating these sub-data out from the abstractions can be hard when using a notion of separation born in an implementation-focused environment. The predicates, however abstract, must carry enough information to undo the separation. There is tension between providing abstractions that allow natural reasoning at the level of the data structure, and easy verification of data structure implementations.

To gain the benefits of local reasoning on abstract structured data, it is helpful to have an equally abstract notion of separation. This "abstract separation" can be designed to break the structure into its natural components without being concerned about implementations or machine heaps. One approach to this "top-down" reasoning is *context logic* [15]. Context logic, like separation logic, is a Hoare logic for local reasoning. However, where separation logic typically works with memory shaped like machine heaps, context logic works with memory shaped like the abstract data being used. For example, when reasoning with lists, abstract lists are stored in the memory directly. The standard technique of *contexts* is used to separate complex data into smaller sub-data, and a novel *separating application* rule allows these sub-data to be reasoned with locally. Importantly, the logic does not consider *how* data might be represented in heap memory. This representation allows the specification of libraries given entirely abstractly. For example, context logic was used to give the first axiomatic semantics to the DOM library [65].

Contexts enable a more abstract notion of locality for structured data. Unfortunately, they lose some of the benefits of separation logic. For one, contexts make little sense for

reasoning about flat heaps, as the machinery of contexts is too heavy handed for simple heap manipulation. This can make reasoning about programs that use both heap memory and abstract data more difficult than it need be. Moreover, the nature of contexts limits certain types of desirable locality. Tree commands such as `appendChild(p, c)` (which moves a tree node `c` to be the child of node `p`) cannot be given truly small axioms, as the contexts requires a larger footprint than the command needs. These larger axioms are both unintuitive and, by being far less local than truly needed, limit reasoning about concurrency.

This thesis seeks the best of both worlds, and arrives at **structural separation logic**. Structural separation logic is designed to mix separation-logic-style heap reasoning with context-logic-style reasoning about structured data. This removes many of the restrictions of context logic, and can obtain truly small footprints for commands like `appendChild`. These small footprints, alongside the restoration of a commutative $*$ operation to reasoning with contexts, enable far more fine-grained *concurrent* reasoning. We also gain natural reasoning about combinations of standard heaps and abstract data. Structural separation logic can create axiomatic specifications for many libraries, including DOM and the POSIX file system.

To develop structural separation logic, we first introduce a standard concurrent programming language, but give it a memory model of *structured heaps*. Structured heaps are like normal machine heaps, but instead of storing heap cells representing the memory used by a specific implementation, store rich structured data directly (see figure 1.1, left). These heaps are not concerned with how a real machine might implement the structured data, instead working directly on abstract representations. They share a basic shape with normal separation logic heaps, allowing low-level implementation focused reasoning to be used alongside the structured data.

We accomplish local reasoning by introducing *abstract heaps*. Abstract heaps allow structured data to be cut-up into sub-data, and enable the sub-data to be "promoted" to the heap level via *abstract allocation*. Abstract allocation is analogous to allocation in imperative programs, but is managed entirely in the logic. It places sub-data in *abstract heap cells*, addressed by *abstract addresses* (figure 1.1, right). These abstract heap cells are disjoint, and so are separated via $*$ in the reasoning. They behave like normal heap cells (albeit cells invisible to the program code), until they are no longer needed. At this point, *abstract deallocation* merges the data back into the value containing the matching *body address*. In this manner, we obtain the high-level reasoning enabled by contexts, and retain the low-level reasoning and natural separation that heaps provide.

We introduce structural separation logic with pedagogical examples, but our goal is

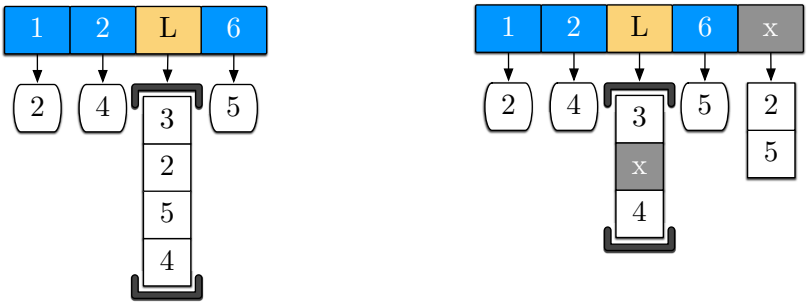


Figure 1.1.: On the left, a structured heap, storing an entire list at address L. On the right, an abstract heap, in which the sub-list $2 \otimes 5$ has been *abstractly allocated* at *abstract address* **x**. Notice the same address then appears as a *body address* in the list at L.

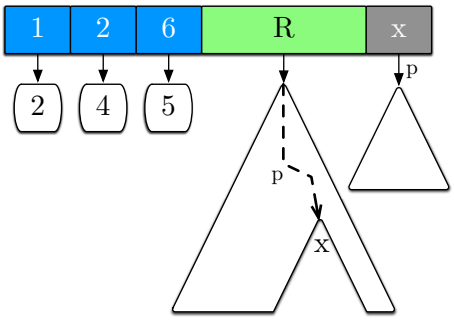


Figure 1.2.: An abstract heap in which abstract cell **x** has promise p.

to give concise axiomatic descriptions of real libraries. This thesis includes two such libraries, the first being the *Document Object Model*. DOM was originally standardised entirely with English prose [71], until the work of Gardner, Smith, Wheelhouse and Zarfaty gave an axiomatic specification using context logic [35, 65]. We revisit this specification in chapter 4 to demonstrate the techniques of structural separation logic, and recast the previous axiomatic semantics using the smaller footprints that our techniques allow. We demonstrate that our specifications for DOM are useful for verifying *client programs* that use the library. By developing a *photo library* program that manages encoded photographic data within an XML file, we show how our techniques can reason about typical uses of the DOM. Moreover, our example requires the use of normal heap memory, demonstrating that we can uniformly handle a mixture of flat heap and structured data reasoning.

Most DOM library commands might be termed "naturally local". The commands work

on sub-trees identified via globally unique *node identifiers*. Each identifier gives a precise handle with which we can extract the data in question, and provides a natural focus for the update. Not all libraries behave this way. Some use global properties of the data structure to select the location of an update, even when that update is focused on a small region. For example, when analysing an element of a list locally, a command may want to know that the element was 5^{th} in the list. The footprint of the command must then contain sufficient data to deduce this fact. Another common example is *paths*. Paths are used in tree manipulating libraries to identify sub-trees. Commands access resource by following the path, then acting on the data at the end of it. These paths can meander around the tree quite freely, but the resultant update is generally local to the sub-tree at the end of the path. Choosing footprints for such commands is thus difficult. The commands need the resource required to follow the path, but will only update a small fraction of it. Requiring that command axioms include all the path resource in addition to the structure needed for the update both complicates the specifications, and drastically limits the number of sub-tree resources that can be disjointly considered simultaneously.

To combat this, we add *promises* to structural separation logic. Promises allow abstract heap cells to retain some information about the structure from which they were allocated. In figure 1.2, there is sub-data at path p within cell R. The abstract cell $\mathbf{x}$ has retained this information, noting that it has been allocated from within a tree at path p. This information has been associated with the data at R as an *obligation* not to change the path to $\mathbf{x}$ until it has been deallocated.

These promises and obligations allow us to give an axiomatic semantics to our second major example, the POSIX file system library. From the large POSIX specification, we distill a core subset that describes the *file system tree*, and the commands that manipulate it. We create a novel model for this subset, and use structural separation logic with promises and obligations to give an axiomatisation. Commands are specified using resource associated with path promises, allowing intuitive axioms that focus on the resource at the *end* of the path. As in DOM, our axiomatisation is useful for proving properties of client programs. We use our logic to reason about an example *software installer*. This program demonstrates the key file system manipulations that software installers must perform. We show that it either completes the installation task correctly (with respect to the file system manipulations), or that no meaningful changes have been made to the file structure. We will also revisit the DOM specifications, and show how promises can further tighten the footprints of some commands.

That structural separation logic can reason about both DOM and POSIX is a strong statement of its usefulness. Both libraries were originally defined only via English speci-

fications, and both are implemented in a variety of ways. They are very widely used by client programs, with our photo library and software installer examples being two typical scenarios. They both use use structured data, but the natural footprints of commands are quite different. In our examples, we see that despite the differences in resource usage, the work required to prove the client programs is very similar. Structural separation logic works at the level of the structured data abstraction, and so allow reasoning about structured data libraries alongside normal, implementation focused, heap reasoning. This is brought out especially in the photo library example, which naturally mixed abstraction layers.

Overall, our contributions can be seen in two parts. First, we offer formal machinery that enables local reasoning with abstract data at a natural level. Our techniques enhance existing methods and reasoning tools, and are built to co-exist with advances in other local Hoare logics. Second, we use our techniques to both specify libraries that use abstract data, and prove clients programs that use the libraries. We have not only enhanced on previous library specifications, but can now provide specifications for libraries not previously analysed in this way. With the techniques and examples of this thesis, we hope to provide library developers with a sound formal system for saying what their libraries do, and programmers with a wider range of feedback methods for software development.

1.1. Contributions

The contributions of this thesis are in five chapters.

Chapter 2 - A Reasoning Framework: *A framework supporting our novel contributions.*

We introduce a reasoning framework, specialising the views system introduced by Dinsdale-Young, Birkedal, Gardner, Parkinson and Yang [22]. Our framework consists of *imperative machines*, described using a simple concurrent programming language, with operational semantics for programs using *structured heaps* as states. We introduce an assertion language that describes sets of structured heaps, but add *instrumentation* to the addresses and values. The choice of instrumentation is left to the user, but is added to enable reasoning when the underlying machine states contain insufficient facts for certain types of proofs (one example being fractional permissions for heap cells). We show how these sets can act as *views*, and so demonstrate a sound Hoare reasoning system.

Chapter 3 - Structural Separation Logic: *A uniform theory for reasoning about combinations of abstract structured data, which allows smaller command axioms than previously possible, as well as the easy mixing of high-level (abstract data) and low-level (standard heap) reasoning.*

Our core contribution, structural separation logic, is a novel program logic for specifying libraries that manipulate structured data, and for reasoning about the programs that use them. We achieve this by splitting data via *structural addresses*. Structural addresses allow data to be broken into smaller *sub-data*, with the address recording the relationships between the data. Sub-data can be *allocated* at a structural *body address*, promoted to a cell within an abstract heap addressed by a structural *abstract address*, treated much like a heap assertion in separation logic, and then *deallocated* to compose again with the body address. Abstract heap cells allow both smaller command axioms than previous work in this area [15], and naturally support a mix of abstract data and normal heap data. This allows hybrid reasoning, combining the abstract reasoning of context logic with the low-level reasoning of separation logic. We give a detailed treatment of reasoning with abstract addresses via simple examples of lists and trees.

Chapter 4 - The Document Object Model: *An case study that revisits previous axiomatic DOM specifications, showing that structural separation logic enables smaller axioms than previous techniques, along with a mix of high- and low-level reasoning.*

We give an axiomatic specification for a subset of the Document Object Model using structural separation logic, building upon previous work [35, 36, 65]. The DOM is a library for tree manipulation, most often used within web browsers as the programming interface between a web page, and script programs that manipulate it. We take the previously identified *featherweight DOM* fragment of the specification [35, 36], which was axiomatised with context logic, and show how structural separation logic can both provide tighter footprints than previous work and integrate reasoning about DOM manipulation with heap manipulation. Following the work of Smith in extending featherweight DOM to specify the entirety of the DOM standard [65], we expect our work on featherweight subset to extend easily to full DOM core level 1. We give an *photo library* example, that shows a typical usage of the DOM library alongside standard heap reasoning. This shows how structural separation logic can naturally handle both flat heap and structured data reasoning simultaneously.

Chapter 5 - Promises: *A general sharing model for abstract structured data, allowing*

local views of abstract data to retain information about the larger environment.

Structural addresses allow rich data to be split into the sub-data that a command requires to function correctly, and the rest of the data that a command does not. However, some libraries use *global* properties to find *local* data. For example, paths in trees are often used to identify the sub-tree on which a command will act. Including all the global data needed to locate sub-data increases the size of command axioms, both decreasing the comprehensibility of axioms and limiting concurrency. (as while data is being used in the footprint of a command, it cannot be used in the footprint of another thread). To combat these problems, we introduce promises. Promises allow sub-data that has been abstractly allocated to retain some information about the context from which it came. For example, an abstract heap cell containing a sub-tree might use a promise to indicate the path at which the sub-tree can be found in the complete tree. Here, we examine *naturally stable* promises, where the facts given by promises cannot be invalidated by any library command. We examine a richer notion in chapter 7.

Chapter 6 - The POSIX File System *(joint work with Gian Nitzk)***:** *A large case study of specifying and structured data library using structural separation logic and promises, using a novel sub-set of the POSIX standard.*

This chapter uses structural separation logic with promises and obligations to give a specification to the POSIX file system library. We identify a subset of POSIX concerned with representing the core *file system tree* and the commands over it. From this, we distill *Featherweight POSIXFS*: a subset of the POSIX file system specification about which we can formally reason. We design an assertion language for POSIXFS based upon trees and heaps, and use structural separation logic to provide axiomatic semantics for our commands. We expect the reasoning we used for this subset to extend naturally to the entire POSIX file systems specification.

Resources used by the file system library are identified with *paths*. We use naturally stable promises to describe sub-data located at the end of paths. This avoids breaking the intuitive notion that a command on file f acts only on file f. We illustrate our approach with an *installer example*. We prove that a simple software installation program either successfully installs a package onto a system, or leaves the file system untouched.

Chapter 7 - Obligations: *A richer sharing model for structured data with, allowing a wider range of libraries to be specified.*

Some useful promises are not naturally stable. For example, the promise that a given element exists at index n within a list is not naturally stable, as the elements in the list prefix up to n could be extended or deleted. We extend promises with a symmetric notion of *obligations*. Just as promises provide information about the wider context of some local data, obligations inhibit certain manipulations of local data to ensure that promises held by the wider context are true. This allows the library to include commands that *could* invalidate promises, as any use such a command will be guarded by the obligations. We use obligations to reduce the size of our DOM specifications.

Taken together, our contributions introduce *library reasoning* for structured data. Our design for structural separation logic allows it to be easily added to many existing - and still to be developed - local reasoning systems. Our goal is that that the data assertions and axioms for a library such as DOM can be easily added to a logic that focuses on a programming language such a C or Java, enriching it with reasoning for the library with minimal effort. We obtain a useful separation of concerns, with structural separation logic focusing on reasoning intuitively about structured data, supported by (and supporting) the broader program logic to which it has been added.

This thesis provides foundational concepts and examples. Building upon this work, we intend to give complete axiomatic semantics for the various levels of DOM, and extend our work on POSIX. We also hope to accelerate reasoning about libraries and languages that have strong relationships, such as joining our DOM reasoning with the JavaScript reasoning of Gardner, Maffeis and Smith [34]. This would allow formal reasoning about web programs in their entirety.

This work will be helped by building on existing separation logic tools. Our abstract heaps easily act as a superset of the symbolic heaps used by tools such as Verifast [45] and the line of automated separation logic tools initiated by Smallfoot [7] (such as SpaceInvader [74] and jStar [25]). We have carried out initial experiments that suggest we can easily integrate with these systems. Resource reasoning about language features can continue to be handled by a tool's normal proof theory. Structured or abstract cells can be handled by a custom engine for structural separation logic, tailored to the domain of the library being used. We can thus develop a system that can "plug-in" to existing tools for separation logic, providing them with easy abstract reasoning for libraries whilst in turn benefiting from their continued advancement.

1.2. Technical background

In 1969, Hoare introduced his eponymous **Hoare logic** [42] for reasoning about imperative programs. Building on previous work by Floyd [29], Hoare logic is based upon *Hoare triples* of the form $\{P\}\ \mathbb{C}\ \{Q\}$. A triple associates *assertions* P and Q with a program $\mathbb{C}$, where P describes the *pre-condition* and Q the *post-condition* of the program. These assertions are first-order logical formulæ that describe the states of the program before and after execution. The meaning of the triple is that, if program $\mathbb{C}$ is run in a state described by P then, if it successfully terminates, the resulting state will be described by Q. This *partial correctness* interpretation says nothing about programs that do not terminate[2]. Hoare's original work did not consider what happens if a program fails, a property of the reasoning he called "conditional correctness".

However, using first-order logic as the assertion language proved inadequate for reasoning about programs that use *heap* (or *dynamically allocated*) memory. This is because programs with such memory are susceptible to *aliasing*, where several variables reference the same piece of memory. With first-order logic, it is difficult to express the effects of commands, as what might appear to be a simple update confined to one heap cell could have unbounded effects via aliasing. For example, the assertion $\mathtt{x} \hookrightarrow 5 \wedge \mathtt{y} \hookrightarrow 6$ describes a global state in which the heap cell at `x` references value 5, and the heap cell at `y` references value 6. If a program performs an update to the cell at `x`, the assertion does not tell us whether the cell at `y` will also be updated, as we do not know if `x` and `y` are aliases.

A similar problem occurs when describing structures. If the assertion $list(l, x)$ describes a linked-list l starting at memory address x, then the assertion $list(l_1, x) \wedge list(l_2, y)$ should intuitively describe two disjoint lists. However, this is not the case. The assertion does not rule out *sharing* between l_1 and l_2 in memory. Updating list l_1 may inadvertently update some of list l_2. Adding anti-aliasing and sharing information proved to be difficult, and ensured that Hoare-style reasoning about heap-manipulating programs languished for many years.

1.2.1. Separation logic

In 2001, O'Hearn, Reynolds and Yang introduced **separation logic** [52], whose assertion language builds on previous work with the logical of bunched implications [53]. They extended first-order logic with linear *heap cell* assertions: the assertion $\mathtt{x} \mapsto 5$ describes a state in which the heap memory cell addressed by variable `x` contains value 5. They also

[2] Hoare simultaneously introduced a *totally correct* interpretation of triples that guarantees termination, but we will not be using this.

added the *separating conjunction*, $*$, which combines heap memory in a disjoint manner. The assertion $\mathtt{x} \mapsto 5 * \mathtt{y} \mapsto 6$ describes a state in which `x` references 5, `y` references 6, and in which `x` and `y` *do not refer to the same heap cell*. The $*$ enforces this disjointness between heap resources. As such, a single heap cell assertion $\mathtt{x} \mapsto 5$ describes *partial* heaps, which may be be extended via $*$. Note that the assertion $\mathtt{x} \mapsto 5 * \mathtt{x} \mapsto 5$ describes no heaps at all, as it is impossible to have two disjoint heap cells that both have address `x`.

The original work on separation logic focused on the *flat heap* (or *RAM*) memory model, a finite partial function between positive integers (acting as "memory addresses") and integers in general (acting as "memory values"). These mappings, known as *heaps*, are comprised of a set of *heap cells*. We will make extensive use of diagrams to demonstrate these heaps. Figure 1.3 shows the heap $1 \mapsto 2 * 2 \mapsto 4 * 6 \mapsto 5$, containing addresses $1, 2$ and 6 with values $2, 4$ and 5 respectively. Any heap can be *separated* into smaller heaps by considering it as the union of the smaller heaps. These small heaps can be composed with the $*$ operator. In the original literature, heap union is described via disjoint function union notation, $\sqcup$. In keeping with the notation of [22], we overload $*$ to act as function union at the model level, in addition to its behavior on assertions.

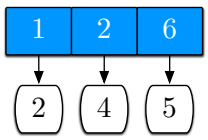


Figure 1.3.: A simple flat heap with three heap cells.

One typical problem in heap-manipulating programs is the access of cells that are not allocated. This causes a *memory fault*, resulting in undefined program behaviour. In focusing on reasoning about heaps, the authors of separation logic recast the interpretation of Hoare triples to ensure that any program proven with separation logic had no memory faults. This **fault-avoiding** interpretation for a triple $\{P\}\ \mathbb{C}\ \{Q\}$ is that, if program $\mathbb{C}$ is run in a state given by P, it will not encounter a memory fault and, if it terminates, it will do so in a state described by Q. Soundness of separation logic was originally proven by O'Hearn and Yang in [75], working with the flat heaps model and a small imperative language with standard operational semantics.

The combination of this interpretation and the resource disjointness given by $*$ founded a school of *local reasoning* for programs. In local reasoning, program proofs are designed to use the minimum amount of resource needed to show correctness. The behaviours of primitive commands are given via axioms with pre-conditions describing only the resource

needed to show that the command does not fault, and its effect on the state. This concept is known as the *footprint* of the command, so called because it states what resource the command "treads on". Two examples for flat heaps are the allocation of a new heap cell, and the update of a heap cell to a new value:

$$\begin{array}{ccc} \{\mathrm{emp}\} & \texttt{x := alloc} & \{\exists \mathrm{X}.(\mathrm{X} \mapsto 0 \wedge \mathtt{x} = \mathrm{X})\} \\ \{\mathtt{x} \mapsto \mathrm{X} \wedge \mathtt{y} = \mathrm{Y}\} & \texttt{[x] := y} & \{\mathtt{x} \mapsto \mathrm{Y}\} \end{array}$$

Notice that the allocation pre-condition requires no resource, since creating a new cell does not alter any others. Changing the value of a heap cell affects only that cell, so the pre-condition for update does not mention any others. These *small axioms* can be extended to account for more memory by adding resource with $*$. As $*$ ensures that the resource is disjoint, we know that the commands could not have accessed it. The *frame rule* exploits this knowledge to add resource to the pre- and post-condition of a triple.

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * R\}\ \mathbb{C}\ \{Q * R\}} \quad \begin{array}{l} \text{If the variables modified by} \\ \mathbb{C} \text{ are not mentioned in } R \end{array}$$

The added resource R is known as the *frame*. Applying the rule is often known as "framing on" or "framing off" resource (depending on the context). The side-condition ensures that variables added by a frame cannot be altered by the program, which would be unsound as the R in the post-condition would not account for the changes. It is possible to pick R that describes resource already described by the pre-condition. The interpretation of triples ensures this is still meaningful, as the pre-condition becomes unsatisfiable by the definition of $*$, and the interpretation is given in terms of states described by the pre-condition. If the pre-condition describes no states, a Hoare triple is valid, but gives no information about the program behaviour.

The elegance of the separation logic frame rule and footprint concept has allowed a flourishing area of automated reasoning tools. By design, each axiom can affect only a small part of a heap. It is therefore often possible for computers to easily determine the effect a command has on an assertion, and so automatically either generate a separation logic proof, or verify an existing one. The first separation logic tool, Smallfoot [7], could prove properties about simple heap manipulating programs using data abstractions such as lists and binary trees. Smallfoot's progeny, tools such as SpaceInvader [74] and jStar [25] are now capable of automatically verifying that large programs are free of memory faults, including the majority of the Linux kernel [24]. There has also been extensive work in proof assistants. One example is Verifast [45] which, when provided with intermediary "ghost" verification instructions, can prove detailed properties about complex programs,

such as implementations of electronic identity cards and device drivers.

Concurrency

Concurrent programs contain more than one thread of execution. A typical problem in concurrent programming with heaps is *data races* where, for example, two threads access the same memory simultaneously. This can leave memory in an inconsistent state, or cause the threads to read invalid data. If we can prove that the code executed by two threads never accesses the same memory, the program can never encounter a data race. The separation afforded by $*$ provides exactly the mechanism to do this, and allows the *parallel rule* of separation logic. This rule, given below, states that if two programs $\mathbb{C}_1$ and $\mathbb{C}_2$ can be proven correct via triples $\{P_1\}\ \mathbb{C}_1\ \{Q_1\}$ and $\{P_2\}\ \mathbb{C}_2\ \{Q_2\}$, it is safe to concurrently execute $\mathbb{C}_1$ and $\mathbb{C}_2$ in states described by $P_1 * P_2$. This type of concurrency is known as *disjoint*, as the two threads are given resource separated via $*$.

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \mid\mid \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

If the variables modified by $\mathbb{C}_i$ are not mentioned in P_j or Q_j for $\{i, j\} = \{1, 2\}$

This rule was proven sound by Brooks in [12]. However, this method does not allow any resource sharing using the flat heap model, as we can never pass a heap cell to two threads. To combat this, Bornat, Calcagno, O'Hearn and Parkinson introduced *permission systems* [10]. The most well-known permission system associates a *fractional permission* with each heap cell. A permission is a rational number in the $(0, 1]$ interval. Heap cells can then themselves be separated by dividing the permission into two, and associating each half with a copy of the cell. This is possible only if $\pi_1, \pi_2 \in (0, 1]$, and $\pi_1 + \pi_2 \leq 1$:

$$\mathtt{x} \overset{\pi_1+\pi_2}{\longmapsto} v \iff \mathtt{x} \overset{\pi_1}{\mapsto} v * \mathtt{x} \overset{\pi_2}{\mapsto} v$$

Each of these fractional `x` heap cells can be passed to a different thread. A thread can update a heap cell if it is contained in its footprint with permission 1. If it has less than permission 1 it can be read, but never written to. Notice this remains *disjoint* concurrency, as the resource passed to the threads is still separated via $*$.

Abstract separation logic

A general framework for program reasoning with separation was not provided until *abstract separation logic* [17], introduced by Calcagno, O'Hearn and Yang. Resource models

in abstract separation logic are *separation algebras*: cancellative, commutative monoids $(H, *, u)$ where H is interpreted as the set of resources and $*$ joins resources. This definition induces a notion of disjointness between data, in that if $h_1 * h_2$ is defined then h_1 and h_2 are disjoint. Abstract separation logic provides a generalised soundness proof for local reasoning with separation algebras, and also shows how *mixed* heaps can be constructed. By taking the cross-product of two separation algebras, one can create heaps involving more than one type of resource. The abstract separation logic approach was the ancestor of several modern formalisms for local reasoning, one of which we will use in this thesis [22].

One non-flat-heap model that has become common is **variables as resource**. Introduced by Bornat, Calcagno and Yang in [10], it treats program variables as linear resource (much like heap cells). This removes the need for the side-condition on the frame and parallel rules. It also improves the uniformity of models for the logic, which no longer need to have separate parts for the heap and variable store. However, it increases the complexity of any axiom that works with variables, and decouples variables from their natural syntactic scoping. Despite this, we will this system to handle variables throughout this thesis, as the uniformity allows us to focus on library reasoning by implementing variables within our structured heaps.

1.2.2. Data abstraction

The assertions of separation logic with flat heaps describe individual allocated memory cells. Structures, such as lists or binary trees, can be described with *inductive predicates*. These predicates provide a representation of the structure using the separation logic memory model, and are somewhat analogous to "implementing" the structure in a language like C. One standard list representation is a singly-linked structure. The list predicate $\mathit{list}(\alpha, x)$ is parameterised by an algebraic list α and the memory location of the first heap cell of the linked list. Algebraic lists are of the form $1 \otimes 2 \otimes 3$, with empty lists denoted $\varnothing$. The standard definition is given in figure 1.4.

The *list* predicate describes complete lists (that is, those linked lists that are null terminated). To describe only part of a list, we can use a *listseg* predicate. The definition of $\mathit{listseg}(\alpha, x, y)$ below describes part of a singly-linked list carrying the contents of algebraic list α, starting at address x and ending at address y.

$$list(\alpha, x) \triangleq \begin{array}{c} \alpha = \varnothing \wedge x = 0 \\ \vee \\ \exists v, \beta, y.\ \alpha = v \otimes \beta \wedge x \mapsto v * (x+1) \mapsto y * list(\beta, y) \end{array}$$

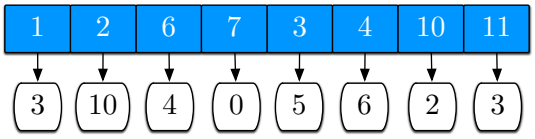


Figure 1.4.: The list predicate for a singly-linked list implementation using binary cells. A diagram of a heap satisfying the predicate $list([3 \otimes 2 \otimes 5 \otimes 4], 1)$ is given below the predicate definition.

$$listseg(\alpha, x, y) \triangleq \begin{array}{c} \alpha = \varnothing \wedge x = y \\ \vee \\ \exists v, \beta, z.\ \alpha = v \otimes \beta \wedge x \mapsto v * (x+1) \mapsto z * listseg(\beta, z, y) \end{array}$$

Programs will often need to gain access to the contents of a structure, such as the individual heap cells that contain the elements inside a $list(\alpha, \mathtt{x})$ instance. To achieve this with inductive predicates requires that they be *rolled* and *unrolled*, replacing the body with the name and vice versa. This rolling and unrolling allows inductive predicates to be separated into more primitive components. The equivalence $\exists y.\ listseg(\alpha, x, y) * list(\beta, y) \iff list(\alpha \otimes \beta, x)$ is justified by unrolling the *list* instance of figure 1.4, then rolling it back into a *listseg* and a *list* instance. This would allow for example, two threads to process the same list concurrently (each thread taking half of the list).

Unfortunately, inductive predicates exhibit poor modularity. As the predicate definition is always available, proofs are tied to the choice of implementation given by the predicate body. Consider a *list library*, that provides client code with functions for interacting with a list. The memory representation of a list should be private to the implementation, so it can be changed. With inductive predicates, nothing prevents the proof of client code from unrolling a predicate and becoming dependent on specific library implementation choices.

To combat this loss of modularity, Parkinson and Bierman introduced *abstract predicates* [57]. Their theory allows the bodies of predicates to be hidden from parts of a proof. The Hoare judgement becomes $\Delta \vdash \{P\}\ \mathbb{C}\ \{Q\}$, where Δ is a *predicate environment* containing a mapping from predicate names to definitions. When its definition is not in scope, an abstract predicate becomes a "black box"; essentially, just a name and some parameters.

The abstract predicate is part of an assertion, but can never be unrolled to give a body. Proof rules are provided to weaken and strengthen the predicate environment. By having client proofs operate on weak environments, and using strengthening to add predicate definitions when entering non-client code, modularity is restored. The library can replace the definition of the predicate at will and, as long as the predicate signature is unchanged, existing client proofs are still valid.

Unfortunately, this modularity comes at the price of locality. Without access to the implementation, an abstract predicate cannot be unrolled into separate components. Recall the *list* in figure 1.4, which we can to obtain a *listseg* for some prefix of the list. Were *list* abstract, clients could not unroll it, and so could not split the list representation. Whilst it is possible to provide axioms regarding abstract predicates, such as $\exists y.\ listseg([\alpha], x, y) * list([\beta], y) \iff list([\alpha \otimes \beta], x)$, each must be justified by appeal to an implementation. The choice of predicate must have sufficient parameters to allow safe abstract joining and splitting, which can restrict implementation choices.

We can see an example of this problem by making the $list(\alpha, x)$ and $listseg(\alpha, x, y)$ predicates abstract. Without the predicate body, clients can only split of lists into smaller *list* and *listseg* instances via axioms. The addresses x and y allow the reconnection of the split data, and work well for linked lists. However, we can conceive of implementations for which these parameters are insufficient to allow reconnection. For example, the list may be backed by a complex structure such a B-tree. In this scenario, splitting a list involves splitting the tree at the implementation level. Rejoining two split trees would require significantly more parameters than are present on the predicates.

Fundamentally, this problem arises as the separation model is chosen for language memory, and not the data model of the abstraction. Facts about the memory model leak across the abstraction boundary, revealing or limiting implementation choices. Thus, the locality exhibited by the predicates is always an expression of the locality of the underlying heap. This problem can be partially mitigated by allowing sharing. The *concurrent abstract predicates* of Dinsdale-Young, Dodds, Gardner, Parkinson and Vafeiadis [23], although designed primarily for concurrent programming, enable such sharing. This system augments the logic with *permissions* and *tokens* over *shared regions*. Shared regions are intuitively groups of heap cells that are given a name and can be shared between threads. They behave additively with respect to separation, so that multiple copies of the same shared region can exist at once (although they all reflect the same underlying heap memory). To avoid inconsistencies, and ensure that concurrent uses of the regions are safe, access to shared regions is guarded via tokens. Updates to the region are performed by actions which are enabled by having certain tokens. The logic guarantees this enforces a sound

protocol for access to the regions.

Shared regions allow a form of reasoning where the separation at the abstract level does not match the separation of an implementation. The example given in [19] is a *set* module, where the abstract predicates are $in(i)$ and $out(i)$. These predicates allow sets to be described in terms of which elements are in the set and which are not. They are separate at the abstract level, allowing assertions of the form $in(1) * out(2) * in(3)$. This describes a set of numbers which certainly contains 1 and 3, certainly does not contain 2, and about which we have no other knowledge. The implementation of the predicates *in* and *out* can be, e.g. a linked list containing *all* the elements contained within the set. The linked list describing the overall set is placed in a shared region, and each predicate is defined in terms of a copy of it, along with tokens allowing the element corresponding to the predicate to be removed (in the *in* case) or added (in the *out* case). Thus, the predicate $in(1)$ has a copy of the region and tokens permitting the entry 1 to be removed from the list. The apparent disjointness at the abstract level is not reflected in the implementation, something the authors termed a *fiction of disjointness*.

Even with concurrent abstract predicates, the theory of separation is still fixed. It must be explained by the underlying language memory model, with the richer separability being enabled by the sharing. *Fictional separation logic* by Jensen and Birkedal [46] attempts to solve this problem in a sequential setting. It allows the assertions of multiple separation algebras to co-exist in a single proof. The basic triples for fictional separation logic are of the form $I.\ \{P\}\ \mathbb{C}\ \{Q\}$, and carry an implicit parameter of Σ, a separation algebra. The I is an *interpretation map*, which takes assertions of the algebra Σ into normal heap assertions. Libraries expose triples to clients with an existentially quantified separation algebra and interpretation map. This gives the library author freedom to expose an interface using a notion of separation appropriate to the module.

This works very well for certain types of abstractions. However, even at the abstract level, the notion of separation is still a commutative operation. To enable fine-grained reasoning, the $*$ operation must allow granular data to be isolated. For rich data, finding a separation algebra that is flexible enough to allow these sub-data to be accessed can be hard. For example, when working with n-ary trees, one often wants to access a specific sub-tree as a separate resource. Trying to find a separation algebra that allows this sub-tree to be separated elegantly, and without disturbing the rest of the tree structure, is difficult, and is one of the contributions of this thesis.

Views

Fictional separation logic allows proofs that expose abstractions via a separation model that is entirely different to that of the standard heap. A similar approach was taken by Dinsdale-Young, Birkedal, Gardner, Parkinson and Yang in *Views* [22]. Sub-titled "compositional reasoning for concurrent programs", the views framework provides a general Hoare-style reasoning system. The key idea of the framework is to split the data representation in two. The states of the underlying machines are described independently from the abstract states used for reasoning. These machine states need not be easily separable nor compositional. Instead, rather than reasoning with machine states, reasoning is performed on *views*. It is these views which are compositional, abstract representations of the states, typically including rich instrumentation.

A view model is a commutative semi-group (VIEWS, $*$), in which every element of VIEWS is a view. Views compose with $*$, but this composition need only make sense with respect to the *instrumentation* being used. Many views represent no machine data at all, instead representing objects used to enable the reasoning (such as the tokens of CAP). The name *view* is apropos, as each object represents a "perspective" on the current states of the machine along with the current "state" of the program reasoning. A view is converted to a set of machine states via a *reification* operation, shown in figure 1.5. Reification transforms a view into the set of machine states it represents. The reification of view p, $\lfloor p \rfloor$, denotes all the machine states described by the view p.

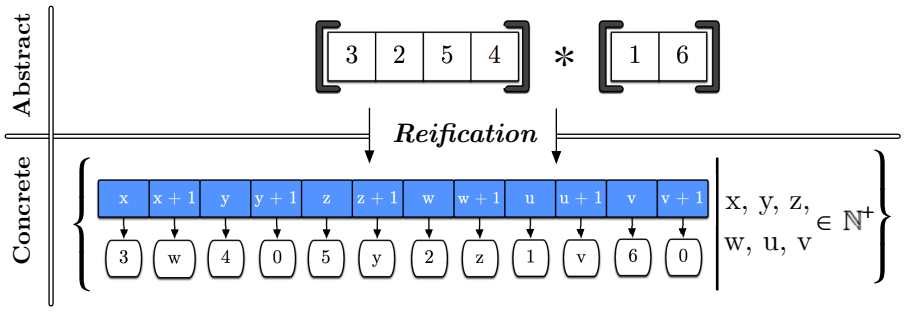


Figure 1.5.: Views uses a two tiered system. The set of views is used to give assertions describing machine states. In this example, each view is a complete list. Reification transforms them into sets of machine states, in this case choosing to represent the abstract lists as singly-linked-lists. Notice the generation of all possible machine states that represent the abstract structure.

Views are used as the pre- and post-conditions of Hoare triples. For each axiom, one must show that the effect of the command is described by the axiom triple, and that the axiom is *stable* (that is, does not disturb other resource). Let views p and q be a choice of pre- and post-condition of some command C, $\{p\}\ \mathrm{C}\ \{q\}$. The choice is good only if, for all possible views r, the effect of the command on all states in $\lfloor p * r \rfloor$ is contained within $\lfloor q * r \rfloor$. The link between triples and commands is thus between the reifications of p and q when extended by *any possible frame* r. Locality and compositionality is "baked in" to the system at every level.

The framework provides a *semantic consequence rule* that allows a view to be altered. The semantic consequence $p \preccurlyeq q$ is justified if, no matter what frame view r is chosen, all the machine states of $\lfloor p * r \rfloor$ are contained within those of $\lfloor q * r \rfloor$. It is informally justified because, from the perspective of the machine, nothing important has changed; all that has occurred is a "view shift". It can be seen as a "ghost update" step, which updates the in the instrumentation. The check that the containment works under all frames ensures that these changes do not conflict with instrumentation on possible frames. Semantic consequence will play a key role in justifying the formalism of abstract heaps in our structural separation logic.

The views framework can represent many disparate reasoning systems, from the original separation logic through to complex modern logics for concurrency such as CAP [23], traditional logics for concurrency such as Rely-Guarantee [47] or Owicki-Gries [55]. It provides a consistent logical framework in which to setup results and prove soundness. We will make extensive use of the views framework throughout this thesis.

1.2.3. Context logic

Meanwhile, in 2005[3], an alternate local reasoning approach for abstract data was being developed. Encouraged by separation logic, but concerned about fine-grained reasoning for structured data, Calcagno, Gardner and Zarfaty introduced **context logic** [15]. Context logic enables local reasoning with data models that that are abstractions of the data being manipulated. The memory models are not heaps, but rather representations of lists, trees, and similar structures.

Contexts are a well known tool in computing theory. There are a variety of types, several of which have been considered as models for context logics. The first presentation was *single-holed* contexts, with typical models being inductively defined data (such as the algebraic list used in figure 1.4), extended with a single *context hole*. The *separating*

[3] Approximately the same time as abstract predicates were introduced.

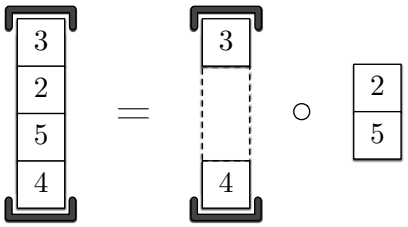


Figure 1.6.: An abstract list $[3 \otimes 2 \otimes 5 \otimes 4]$ decomposed via separating application into the context $[3 \otimes - \otimes 4]$ and sub-list $2 \otimes 5$, representing the entire list as $[3 \otimes - \otimes 4] \circ (2 \otimes 5)$.

application operator $\circ$ allows data to be pulled apart, leaving a context with a hole and some *sub-data*, as in figure 1.6.

Using contexts to represent data enables different forms of locality than the separation concepts seen so far. Most importantly, it is *top-down*, allowing us to use a data model that matches the abstraction level of the data it is manipulating. Freed from the need to provide a *, the separating application can be non-commutative. There is a difference between a context (which has a hole) and data (which fits in a hole). The relationships between data and sub-data can be recorded via the context hole, without consideration for a representation within some machine memory. This is reflected in the frame rule of context logic, which just adds a context over an entire datum.

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{R \circ P\}\ \mathbb{C}\ \{R \circ Q\}} \quad \text{If the variables modified by } \mathbb{C} \text{ are not mentioned in } R$$

Context logic was used by Gardner, Smith, Wheelhouse and Zafarty to give an axiomatic specification to a subset of the *Document Object Model Library Core Level 1* standard [35, 36]. Smith later extended this subset to the entire standard. The commands of DOM manipulate a tree structure that is described abstractly. Contexts could give local footprints to many of the commands, but certain situations (such as `appendChild`) still required more resource that would be expected.

Moreover, using single-holed contexts also restricts some types of locality. Context applications can be nested in an associative fashion, but a single datum cannot contain more than one hole. The lack of commutatively mean there is no natural analogue of the parallel rule from separation logic. The first of these problems was addressed by Calcagno, Dinsdale-Young and Gardner in [14]. They extended the single-holed models to support *multiple labeled holes*. These multi-holed contexts were developed for technical results

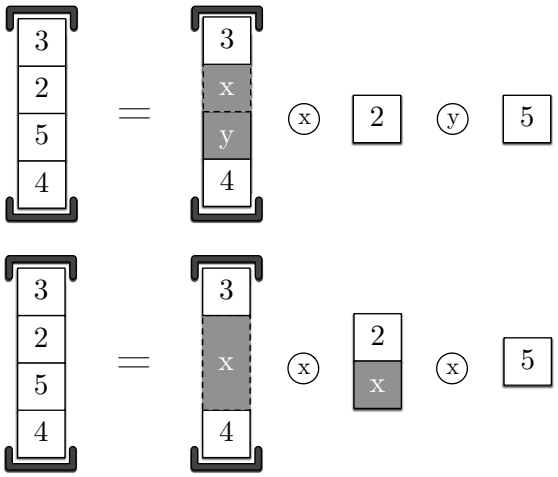


Figure 1.7.: Two multi-holed contexts that describe the same list. The separating applications bind the holes **x** and **y** within the lists. Two contexts may use the same hole as long as composition is unambiguous.

regarding adjoint elimination, but provide for richer splittings of data, as in figure 1.7.

By naming holes, composition gains *quazi-commutativity*, such that in cases like the top of figure 1.7 the order of composing 2 and 5 into the larger list is irrelevant. However, composition is still not commutative. There is a difference between $P \circledast_{\mathbf{x}} Q$ and $Q \circledast_{\mathbf{x}} P$, as P *must* have an **x** hole, but Q need not. Composition remains a notion of "filling holes", with holes in contexts bound by the composition operation, so that the bottom result of figure 1.7 is allowable.

Multi-holed contexts allow data analysis at a finer level than single-holed. Consider a command `appendChild(p, c)`, acting on a tree. It moves the sub-tree with top node `c` to be the last child of node `p`.

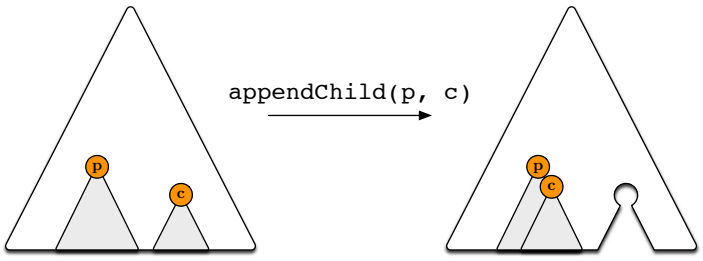


Figure 1.8.: The action of the command `appendChild(p, c)`.

This command can fail, in a manner analogous to a memory fault, if the node identified by `p` or `c` does not exist in a tree. However, even assuming that both nodes exist in a tree, this command can still fail if the node `c` is an ancestor of node `p`. Were this possible, it would create a "tree' where `c` is a sub-tree of itself.

To axiomatise this command, we need to describe the two safe cases: one where the node `p` is an ancestor of the node `c`, and one where the two nodes have no parent/child relationship. To capture the two cases using contexts, we must find a single splitting of the data that describes both. Such a case can be found by taking a *covering context*: one that captures the smallest tree containing both `p` and `c`. This allows the pre-condition to be expressed as $(C \circledast \mathtt{p}[t]) \circledcirc{y} \mathtt{c}[t']$, one case of which is below. This would not be possible with single-holed contexts, as one possible splitting here is that C contains both the $\mathbf{x}$ and $\mathbf{y}$ hole.

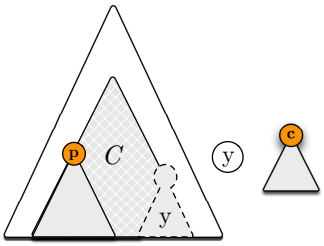


Figure 1.9.: To give a single description of the states safe for `appendChild(p, c)` with multi-holed contexts, a *covering context* must be used to capture the entire sub-tree containing both node `p` and `c`. In this case, the covering context C is for the case where `c` is not a child of `p`.

Multi-holed contexts still have limitations. The axiom for the pre-condition of `appendChild` would ideally be in terms of the node at `p` and the sub-tree at `c`. The covering context is not touched by the command, but must be in the footprint due to the nature of contexts. Moreover, the concurrency rule is still inapplicable. These problems occur because of the non-commutative nature of separating application, which acts as a binder to the multi-holed context to its left.

1.3. Introducing structural separation logic

We now give a high-level introduction to the concepts of **structural separation logic**, the main technical contribution of this thesis. Structural separation logic fuses the natural

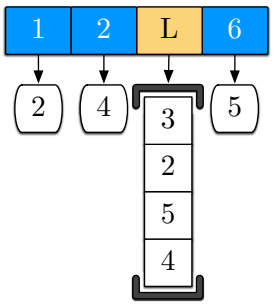


Figure 1.10.: A structured heap, with regular heap cells at addresses $1, 2$ and 6, and a list heap cell at address L containing the list $[3 \otimes 2 \otimes 5 \otimes 3]$.

heap reasoning of separation logic with the natural abstract reasoning of contexts, and does so without complicating assertions with additional machinery. Contexts and heap cells are freely mixed in assertions, and are connected with a standard associative and commutative $*$ conjunction. Abstract and flat heap data can be reasoned about using the same techniques. We can give tighter axioms to library specifications and more compact proofs of code that use them.

We achieve this by working with *structured heaps* at the operational level, and *abstract heaps* at the reasoning level. Structured heaps allow us to avoid considering the implementation of data structures. They are similar to the flat heaps of separation logic but, rather than consisting of heap cells with simple values like integers, are of a mix of cells storing flat and *rich* values. For example, alongside standard values, a cell can contain an entire list (see figure 1.10).

A model of *imperative machines* is given that treats these structured cells like any other memory. Commands reference the data by address, and perform imperative update within the structured value, changing it atomically from one structured datum to another. The use of rich values is optional, so that structured heaps behave as standard heaps if needed.

A key contribution is a reasoning model for structured data using *abstract heaps* and *structural addresses*. Abstract heaps are like structured heaps, but contain *abstract heap cells* storing sub-data from within some structure. Structural addresses enable this in two ways. Firstly, data can contain a structural *body address*, giving a location in which sub-data can be applied. Secondly, abstract heaps cells are addressed via a structural *abstract address*, linking the sub-data to the body address from which it came. Abstract heap cells are created by *abstractly allocating* sub-data from within some structure into a freshly addressed abstract cell. They can be destroyed by *abstractly deallocating* an abstract cell

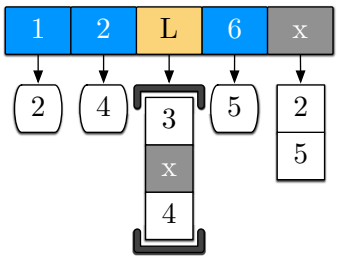


Figure 1.11.: An abstract heap, where the sub-list $2 \otimes 5$ has been *abstractly allocated* at *structural address* $\mathbf{x}$

back into the matching body address. In figure 1.11, the list of figure 1.10 on the preceding page has been split by abstract allocation to create a new abstract heap cell at address $\mathbf{x}$. When this heap cell is no longer needed, it will be deallocated, restoring the complete list.

In the reasoning, abstract heap cells allow sub-data to be treated exactly like any other heap cell. When needed, a rule within the proof theory performs the "allocation". Axioms for commands on structured data then work with the smallest abstract heap cells that contain their footprint. Once the operation is complete, *abstract deallocation* removes the abstract cell, and recombines the data. Abstract allocation and deallocation are performed by directly updating the assertions, and do not requiring additional machinery in the assertion language.

Such updates to the model are enabled by our adoption of the *views framework* [22]. As discussed in section 1.2.2, views provides us with local and compositional reasoning. However, the key facility the framework gives us is *reification*, and with that, *semantic consequence*. Reification allows the abstract heaps we use for Hoare reasoning to be interpreted as structured heaps. Reification removes abstract heap cells and collapses the data they contain back into the corresponding body address. Moreover, it also handles *partial* abstract heaps. It is possible to construct abstract heaps that are missing the abstract heap cells needed to represent complete data. Reification provides these by *completing* the data in every possible way. Our use of reification is illustrated in figure 1.12.

The semantic consequence relation of views enables abstract allocation and deallocation. Note that our structural addresses are a form of instrumentation on heaps. By designing these addresses so that two abstract heaps reify to the same structured heaps if they differ only by uses of abstract allocation and deallocation, we can update the splitting of the heap using only semantic consequence steps.

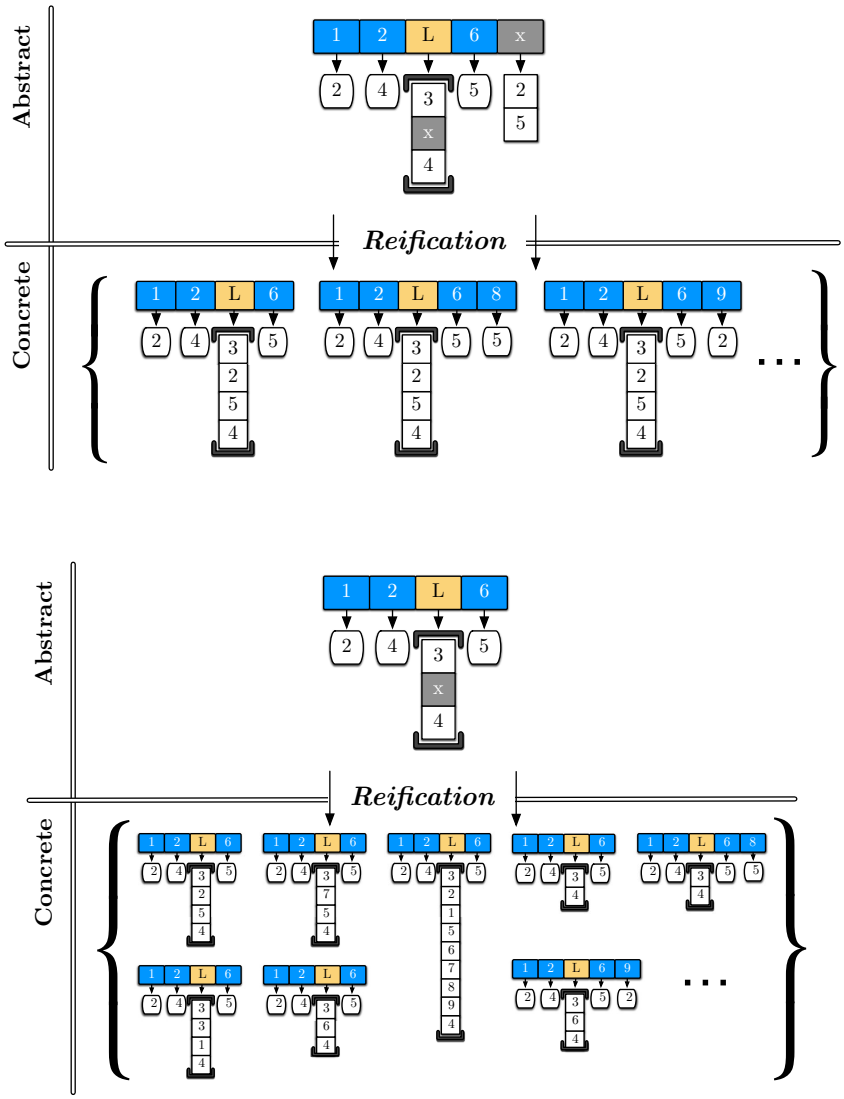


Figure 1.12.: If all abstract heap cells are present, reification rejoins them. Abstract heaps are also *completed* in all possible ways, to account for possible missing data. The results are thus sets of machine heaps.

There has been previous work with similar goals to structural separation logic. Wheelhouse's *segment logic* [38, 70] restored an associative $*$ operation to working with contexts. He achieved this by splitting data into a set of *segments*, where each segment was either *rooted* or labelled with a context hole. Models of assertions were sets of segments, which were separated by $*$. These models are somewhat similar to the abstract heaps of structural separation logic.

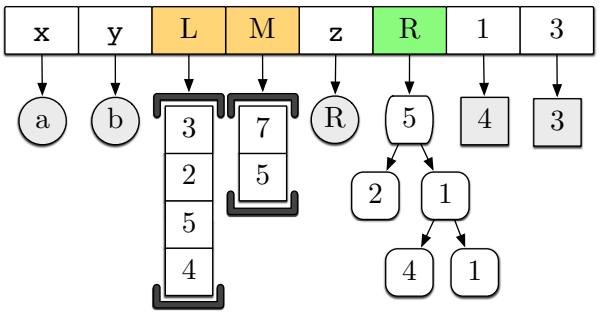
Unfortunately, reasoning with segment logic can be somewhat cumbersome. The structure of the models was not simple heaps, but rather *segment algebras*. To ensure $*$ was associative, *binders* had to be introduced into assertions, so that a segment $\mathrm{L} \mapsto [3 \otimes 2 \otimes 5 \otimes 4]$ could not directly be represented as $\mathrm{L} \mapsto [3 \otimes \alpha \otimes 5 \otimes 4] * \alpha \mapsto 2$. Instead, *revelation* (from the ambient calculus [18]) and *freshness quantification* was used, so that $\mathrm{L} \mapsto [3 \otimes 2 \otimes 5 \otimes 4] = \nu\alpha.\ \alpha \circledR (\mathrm{L} \mapsto [3 \otimes \alpha \otimes 5 \otimes 4] * \alpha \mapsto 2)$. By adding these to the assertion language, new rules were needed in the reasoning to manage them.

The combination of these bindings, and the non-heap model, means that the local reasoning provided by segment logic comes with rather a lot of baggage. With the advent of the views philosophy, the key ideas behind segment logic were refined, and formed the genesis of structural separation logic. The key advantages and disadvantages of each techniques reflect the technology available at their creation.

The design of structural and abstract heaps allows several data models to be combined together. Normal "low-level" separation logic reasoning can be performed alongside the use of a library representing, e.g. lists. Moreover, as the complexity of the reasoning is hidden behind reification operations, the assertion language remains simple. Abstract allocation and deallocation occur via a proof rule, justified by the properties of reification. These results mean that structural separation logic axioms for a library could easily be added to most other separation logic reasoning systems, from the original work of O'Hearn [63], through Parkinson's separation logic for Java [56] and even Smith's work on JavaScript [34]. The theory "bolts on" the side, giving extra expressively with minimal effort. We also expect that integration with tools such as Verifast [45] and JStar [25] will be straightforward. Our abstract heaps can exist alongside the symbolic heaps used by the tools, so as we develop automation for structural separation logic, we can extend and cooperate with the development of automation in separation logic.

2. A reasoning framework

This chapter presents a local reasoning framework for programs that manipulate *structured heaps*, providing a foundation for the rest of this thesis. Like a standard heap, a structured heap associates *addresses* with *values*. Unlike a standard heap, the values can have rich structure. In the below example, the *addresses* $\mathtt{x}, \mathtt{y}, \mathtt{z}, 1$ and 3 map to to simple flat values, but the addresses L and M map to complete lists and R maps to a complete tree.



Our framework gives reasoning for *imperative machines*. Imperative machines provide a model for programming built from structured heaps, a concurrent programming language, and an operational semantics. They formalise structured heaps as sets of *addressed values*. Their definition is flexible, allowing many types of addresses, values and commands to be used. This enables the representation of programming *libraries*.

We give a program reasoning system based on *instrumented structured heaps*. Like structured heaps, these are formed of addresses and values, but are decorated with additional *instrumentation* to facilitate reasoning. Instrumented states are collected into sets called *views*, which can be converted into sets of structured heaps by a process of *reification*. This reification interprets the instrumentation, and allows views to represent possible states at each point in program execution. The views used by us are those of the Views framework [22], discussed in section 1.2.2. They allow us to build a *semantic reasoning system*, and construct Hoare triples that specify interesting program properties. We extend the work of [22] with a syntactic *assertion language* that describes views, and can be used as the pre- and post-conditions of a local Hoare calculus.

To aid understanding, the examples are designed to link our framework to well known local reasoning techniques. We show how to reason about programs manipulating *variables as resource*, *flat heaps*, and *resource with permissions*. Variables as resource is a commonly used technique for managing variables in separation logic style proofs [11]. Our term *flat heaps* refers to the original work on separation logic [63]. Augmenting resources with fractional permissions [10] has become a standard method for reasoning about shared resource. The full generality of our framework will be used to in upcoming chapters.

We will not use the full generality of structured heaps in this chapter, focusing on introducing the concepts of the framework via flat heaps. The purpose of this thesis is to extend the reasoning about these well understood heaps to reasoning about libraries that manipulate highly structured data. We will only hint at these uses here, but in future chapters, structured heaps will allow us to give imperative machines that directly manipulate this rich data. We will use them as the basis of structural separation logic in chapter 3, DOM in chapter 4 and POSIX file-systems in chapter 6. The reasoning system and soundness proofs presented here will be reused without change on these more complex values.

2.1. Imperative machines

Imperative machines are abstract computational systems consisting of three parts: a *state*, a *program*, and an *operational semantics*. Their definition is parametric in several places, allowing different types of addresses and values to be used.

2.1.1. Structured heaps

Our imperative machines have state. These states are *structured heaps*, in which data is represented by *addressed values*.

Parameter 1 (Machine addresses and values)**.** Assume a non-empty countable set of **machine addresses** MachineAddrs, ranged over by $a, a_1, \cdots, a_n$. Assume also a non-empty countable set of **machine values** MachineVals, ranged over by $v, v_1, \cdots, v_n$.

Neither the set of addresses nor values need be infinite. There also need be no relationship between the sets MachineAddrs and MachineVals.

Example 1 (Machine addresses and values)**.** The following are some example choices for machine addresses and values.

Program variables: Let MachineAddrs be a set of program variable names, and MachineVals be the integers $\mathbb{Z}$ and booleans $\{\mathbf{true}, \mathbf{false}\}$. This is one choice of addresses and values for *program variables* (e.g. `x = 5`). Notice that MachineAddrs $\not\subset$ MachineVals, as we do not store variable names within variables, nor do we consider integers or booleans as variable names. This relationship is different from separation logic, where addresses are contained within values.

Flat heaps: Let MachineAddrs $= \mathbb{N}^+$, and MachineVals $= \mathbb{N}$. These choices are suitable for flat heaps, the style of model used in the original separation logic papers [63]. Notice both that MachineAddrs is infinite in size, and that MachineVals $\subset$ MachineAddrs.

Trees: Let MachineAddrs $= \{\top\}$ (a single address), and MachineVals be a set of trees. If the tree nodes have distinct identifiers, are given names, and have arbitrarily many children, then they are suitable for constructing *DOM trees*. If tree nodes have no identifiers, but have names distinct between siblings and arbitrarily many children, then these choices are suitable for a simple *file-system directory structure*. We will investigate systems of these types in chapters 4 and 6 respectively.

Flat heap and variables: We can combine multiple address/value choices together. For example, MachineAddrs could be variable names *and* the natural numbers. The set MachineVals would then be $\mathbb{Z} \cup \{\mathbf{true}, \mathbf{false}\}$. This gives us a single heap which can store both variables and flat values. We refer to these as *primitive heaps*, and they will be our primary example in this chapter.

Addresses and values are paired together to form *heap cells*. The state of an imperative machine is a set of heap cells, where each address unique. We call this a *structured heap*, or just *heap* if the context is clear. We will often give diagrams to structured heaps. An example for primitive heaps is given in figure 2.1.

It is sometimes convenient to prevent heaps from containing certain heap cells. One example of this is a type system for variables, where some variables must store values of a certain type. Another is when combing two sets of addresses, as in the primitive

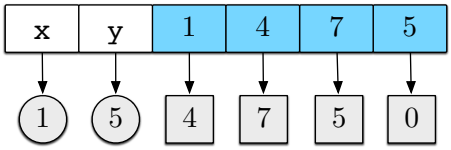


Figure 2.1.: Example rendering of a primitive heap, containing variable heap cells $\mathtt{x} \mapsto 1$ and $\mathtt{y} \mapsto 5$ and flat heap cells $1 \mapsto 4, 4 \mapsto 7, 7 \mapsto 5$ and $5 \mapsto 0$.

heaps above, where we want to ensure that flat heap cells only contain values from $\mathbb{N}$. The definition of structured heaps accounts for these possibilities by being a subset of all definable structured heaps.

Parameter 2 (Structured heaps)**.** Given a set of machine addresses MachineAddrs and machine values MachineVals (parameter 1), assume a set of **structured heaps** StructHeaps, ranged over by $s_1, \cdots, s_n$, with type:

$$\textsc{StructHeaps} \subseteq \{s \mid s : \textsc{MachineAddrs} \xrightarrow{\text{fin}} \textsc{MachineVals}\}$$

Imperative machines allow programs that cause machines to "go wrong". The notion of "wrong" is situation specific, but is a common feature of languages and machines. For example, the commands of the flat heap example will "go wrong" when instructed to access an address not present in the structured heap. The singleton heap $5 \mapsto 3$ contains no 4 address. When the command $\mathtt{x} := [4]$ is run, it will try to access address 4, and fail. Another example is the DOM tree library. Each DOM node in the tree has a unique identifier. Attempting to access an identifier not present in the tree will fail.

Whenever a machine attempts to perform an impossible operation, it enters a *fault state*. A choice of commands may have several ways of faulting. For example, DOM commands will also fail if instructed to create a node with a name containing certain reserved characters. We are not interested in the specific modes of failure, so there is only one fault state.

Definition 1 (Fault state)**.** Given a set of structured heaps StructHeaps (parameter 2), let $\lightning$ be the single **fault state**, where $\lightning \notin \textsc{StructHeaps}$.

Structured heaps plus the fault state form the set of *outcome states*, representing the final result of a program.

Definition 2 (Outcome states)**.** Given a set of structured heaps StructHeaps (parameter 2) and a fault state $\lightning$ (definition 1), the set of **outcome states** is defined as:

$$\text{Outcomes} \triangleq \text{StructHeaps} \cup \{\lightning\}$$

Our examples will almost all use program variables. We define a simple set of variables and values.

Definition 3 (Program variables)**.** Assume a set of program variables $\text{PVars} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{a}, \mathtt{i}, \mathtt{foo}, \cdots\}$. Assume also a set of program values $\text{PVals} = \mathbb{Z} \cup \{\textbf{true}, \textbf{false}\}$.

Example 2 (Variables as resource)**.** By selecting the machine addresses and values (parameter 1) as PVars and PVals respectively, we can define the structured heaps (parameter 2) for variables as resource [11] as:

$$\text{StructHeaps} \triangleq \{s \mid s : \text{PVars} \xrightarrow{\text{fin}} \text{PVals}\}$$

Example 3 (Flat heaps)**.** To model flat heaps, select $\text{MachineAddrs} = \mathbb{N}^+$ and $\text{MachineVals} = \mathbb{N}$. The structured heaps are:

$$\text{StructHeaps} \triangleq \{s \mid s : \mathbb{N}^+ \xrightarrow{\text{fin}} \mathbb{N}\}$$

Example 4 (Primitive heaps)**.** Flat heaps alone are not useful, as programs cannot reference heap addresses. It is standard to use program variables to store these addresses. We thus combine the variable store and flat heap examples, calling the result **primitive heaps**.

We use program variables and values from definition 3, and so let $\text{MachineAddrs} = \text{PVars} \cup \mathbb{N}^+$, and $\text{MachineVals} = \text{PVals}$ (recalling that $\mathbb{Z} \subset \text{PVals}$). The heap definition is:

$$\textsc{StructHeaps} \triangleq \left\{ s \;\middle|\; \begin{array}{l} s : \textsc{PVars} \cup \mathbb{N}^+ \xrightarrow{\text{fin}} \textsc{PVals}, \\ \forall a \in \text{dom}(s).\ a \in \mathbb{N}^+ \implies s(a) \in \mathbb{N} \end{array} \right\}$$

Notice that flat heap addresses are only associated with natural numbers, preventing heaps from mapping flat heap addresses to negative numbers or booleans.

Comment 1. In many presentations of similar work, machine states are often split in two, with the heap and variable store being different. Heaps are treated as linear resource, variables as "pure" facts. I will use only linear resource, adopting the variables as resource approach from [11]. This allows the imperative machine states to be of a uniform type, and so simplifies the reasoning. It is possible to use a separate variable store with the work in this thesis but, as handling variables is orthogonal to my library reasoning programme, I opt for presentational simplicity.

2.1.2. Programming language

Imperative machines programs are written using a simple concurrent WHILE language. Programs consist of basic operations (the atomic commands and skip), flow control constructs (sequencing, conditional choice and looping), and parallel composition.

Definition 4 (Basic programming language)**.** The **programs** of the basic programming language Programs, ranged over by $\mathbb{C}, \mathbb{C}_1, \cdots, \mathbb{C}_n$, are defined by induction as follows: for all b in BoolExprs and c ∈ AtomicCmds (to be given by parameters 3 and 5 respectively)

$\mathbb{C} ::=$	c	Atomic Commands
	| `skip`	No operation
	| $\mathbb{C}_1$; $\mathbb{C}_2$	Sequencing
	| `if (`b`)` $\mathbb{C}_1$ `else` $\mathbb{C}_2$	Conditional choice
	| `while (`b`)` $\mathbb{C}$	Conditional loop
	| $\mathbb{C}_1 \,\|\|\, \mathbb{C}_2$	Parallel composition

The `if` and `while` commands are parametrised by *boolean expressions*. These expressions are evaluated to determine which branch of a conditional choice to take, and to

guard loops. In most situations, the boolean expressions will be a subset of some richer expression language. However, we only mandate the existence of boolean expressions, as they are all we need to give an operational semantics to the language.

Parameter 3 (Boolean expressions)**.** Assume a set of **boolean expressions** BoolExprs, ranged over by $\textsc{b}_1, \cdots, \textsc{b}_n$.

A boolean expression is evaluated on a structured heap, resulting in a boolean truth value or a fault. Faults indicate a failure of evaluation. The heaps in which boolean expression evaluation faults are dependent on the choice of expression language. Typical reasons for faulting are an expression referencing an unassigned variable or type errors such as performing arithmetic on strings but, as discussed, we consider them all be a single fault type. We make the assumption that expression evaluation never alters the state, and is deterministic.

Parameter 4 (Boolean expression evaluation)**.** Given a set of boolean expressions BoolExprs (parameter 3) and structured heaps StructHeaps (parameter 2), assume a **boolean expression evaluation** function $[\![\cdot]\!](\cdot) : \textsc{BoolExprs} \to \textsc{StructHeaps} \to \{\textbf{true}, \textbf{false}, \lightning\}$. The evaluation of boolean expression $\textsc{b}$ in state s is written as $[\![\textsc{b}]\!](s)$.

Example 5 (Expressions for primitive heaps)**.** We define boolean expressions for our primitive heap example in terms of **basic expressions**. Basic expressions are either a variable lookup (e.g. $\mathtt{x}$), a literal value (e.g. **true** or 5), or the sum of two other expressions. Formally, the **basic program expressions** Exprs, ranged over by $e, e_1, \cdots, e_n$, are defined inductively as: for all $\mathtt{x} \in \textsc{PVars}, v \in \textsc{PVals}$

$$\begin{array}{rrr} e ::= & v & \text{Literal value} \\ & \mid \mathtt{x} & \text{Program variable} \\ & \mid e_1 + e_2 & \text{Integer sum} \end{array}$$

Literal value expressions evaluate to the literal itself. The evaluation of a program variable expression just looks up the variable in the heap, and faults if it is not present. Integer sums evaluate to the summation of evaluating the operands (faulting if the results are not integers). Formally, using the expressions Exprs, primitive heaps

StructHeaps (example 4) and program values PVals (example 2), the evaluation function for basic expressions $(\!|\cdot|\!)(\cdot) : \text{Exprs} \to \text{StructHeaps} \to \text{PVals} \cup \{\lightning\}$ is defined as:

$$
\begin{aligned}
(\!|v|\!)(s) &\triangleq v \\
(\!|\mathtt{x}|\!)(s) &\triangleq \begin{cases} s(\mathtt{x}) & \text{if } \mathtt{x} \in \mathrm{dom}(s) \\ \lightning & \text{otherwise} \end{cases} \\
(\!|e_1 + e_2|\!)(s) &\triangleq \begin{cases} (\!|e_1|\!)(s) + (\!|e_2|\!)(s) & \text{if } (\!|e_1|\!)(s), (\!|e_2|\!)(s) \in \mathbb{Z} \\ \lightning & \text{otherwise} \end{cases}
\end{aligned}
$$

Boolean expressions are then defined in terms of these basic expressions. The primitive boolean expression is just a basic expression, which will evaluate to fault unless the basic expression evaluates to a boolean value. The others are standard, but note that equality comparison is between boolean expressions rather than basic expressions. This allows programs to include idiomatic boolean expressions like $\neg\text{B} = \mathbf{true}$. Formally, using the expressions Exprs, the **boolean expressions** for primitive heaps, BoolExprs, are defined inductively as: for all $e, e_1, e_2 \in \text{Exprs}$

$$
\begin{aligned}
\text{B} ::= \quad & \mid e && \text{Basic expression} \\
& \mid \neg\text{B} && \text{Negation} \\
& \mid \text{B}_1 = \text{B}_2 && \text{Equality} \\
& \mid e_1 < e_2 && \text{Inequality}
\end{aligned}
$$

The evaluation for the boolean expressions is standard. Type checking is performed, with faults occurring when the types are incorrect. Formally, the evaluation function for boolean expressions $[\![\cdot]\!](\cdot) : \text{BoolExprs} \to \text{StructHeaps} \to \{\mathbf{true}, \mathbf{false}, \lightning\}$ is defined as:

$$
\begin{aligned}
[\![e]\!](s) &\triangleq \begin{cases} (\!|e|\!)(s) & \text{if } (\!|e|\!)(s) \in \{\mathbf{true}, \mathbf{false}\} \\ \lightning & \text{otherwise} \end{cases} \\
[\![\neg \mathrm{B}]\!](s) &\triangleq \begin{cases} \neg [\![\mathrm{B}]\!](s) & \text{if } [\![\mathrm{B}]\!](s) \in \{\mathbf{true}, \mathbf{false}\} \\ \lightning & \text{otherwise} \end{cases} \\
[\![\mathrm{B}_1 = \mathrm{B}_2]\!](s) &\triangleq \begin{cases} [\![\mathrm{B}_1]\!](s) = [\![\mathrm{B}_2]\!](s) & \text{if } [\![\mathrm{B}_1]\!](s), [\![\mathrm{B}_2]\!](s) \neq \lightning \\ \lightning & \text{otherwise} \end{cases} \\
[\![e_1 < e_2]\!](s) &\triangleq \begin{cases} (\!|e_1|\!)(s) < (\!|e_2|\!)(s) & \text{if } (\!|e_1|\!)(s), (\!|e_2|\!)(s) \in \mathbb{Z} \\ \lightning & \text{otherwise} \end{cases}
\end{aligned}
$$

The basic programming language of definition 4 is also parameterised by *atomic commands*. These commands mutate the heaps, and are *atomic* as they do so in an instantaneous and uninterruptible fashion. Command choices range from simple variable assignment, through heap cell allocation and deallocation (in flat heaps), to commands found in complex libraries such as DOM and POSIX.

Parameter 5 (Atomic commands)**.** Assume a set of **atomic commands** AtomicCmds, ranged over by $\mathrm{C}, \mathrm{C}_1, \cdots, \mathrm{C}_n$.

Each atomic command is associated with an *action* that defines its behaviour. An action is a total function, transforming input heaps into *sets* of output heaps. Commands can thus be non-deterministic.

Parameter 6 (Atomic command actions)**.** Given a set of outcome states (definition 2), for each atomic command $\mathrm{C} \in$ AtomicCmds (parameter 5), assume an atomic command action $[\!|\mathrm{C}|\!](\cdot) : \text{StructHeaps} \to \mathcal{P}(\text{Outcomes})$.

It is occasionally desirable that some commands do not fault, yet never terminate when executed in a state (for example, concurrency primitives such as `lock`). This is modeled by taking the atomic action in such states to be the empty set.

Example 6 (Commands and actions for primitive heaps)**.** We give a standard set of commands for interacting with flat heaps and variables: assignments of expressions

to variables; allocation and deallocation of flat heap cells; assignment to flat heap cells; and the dereferencing of flat heap cells. Formally, given program variables PVars (definition 3) and expressions Exprs (example 5), the commands are: for all $\mathtt{x} \in \text{PVars}$ and $e, e_1, e_2 \in \text{Exprs}$

`x := `e	Variable assignment
`x := alloc()`	Cell allocation
`free(`e`)`	Cell deallocation
`[`e_1`] := `e_2	Cell update
`x := [`e`]`	Cell dereference

Given that $f[d \mapsto c]$ denotes a function that behaves as f, except that d maps to c, the atomic command actions for the commands are: for all $\mathtt{x} \in \text{PVars}$ and $e, e_1, e_2 \in \text{Exprs}$

$$
\begin{aligned}
[\!\langle \mathtt{x := } e \rangle\!](s) &\triangleq \begin{cases} \{s[\mathtt{x} \mapsto (\!|e|\!)(s)]\} & \text{if } (\!|e|\!)(s) \neq \lightning \\ \{\lightning\} & \text{otherwise} \end{cases} \\
[\!\langle \mathtt{x := alloc()} \rangle\!](s) &\triangleq \{s[\mathtt{x} \mapsto i, i \mapsto v] \mid i \in (\mathbb{N}^+ \setminus \mathrm{dom}(s)), v \in \mathbb{N}\} \\
[\!\langle \mathtt{free(}e\mathtt{)} \rangle\!](s) &\triangleq \begin{cases} \{s|_{\mathrm{dom}(s) \setminus \{(\!|e|\!)(s)\}}\} & \text{if } (\!|e|\!)(s) \in \mathrm{dom}(s) \\ \{\lightning\} & \text{otherwise} \end{cases} \\
[\!\langle \mathtt{[}e_1\mathtt{] := } e_2 \rangle\!](s) &\triangleq \begin{cases} \{s[(\!|e_1|\!)(s) \mapsto (\!|e_2|\!)(s)]\} & \text{if } \begin{array}{l} (\!|e_1|\!)(s) \in \mathrm{dom}(s), \\ (\!|e_2|\!)(s) \neq \lightning \end{array} \\ \{\lightning\} & \text{otherwise} \end{cases} \\
[\!\langle \mathtt{x := [}e\mathtt{]} \rangle\!](s) &\triangleq \begin{cases} \{s[\mathtt{x} \mapsto s((\!|e|\!)(s))]\} & \text{if } (\!|e|\!)(s) \in \mathrm{dom}(s) \\ \{\lightning\} & \text{otherwise} \end{cases}
\end{aligned}
$$

The assignment action is terminating and deterministic (as expression evaluation is deterministic). Faults in evaluating the expression become faults of the command action.

The action for allocation adds a new cell to the heap at a fresh address. This address is assigned to the variable, and is chosen non-deterministically from those not already used by the heap. As there are infinitely many addresses to pick (recall the heap domain is $\mathbb{N}^+$), this command never faults. The contents of the new heap cell is also

non-deterministically selected from all possible values. This models the behaviour of heap cell allocation in a 'C'-like language that has no memory constraints.

The deallocation action removes a cell from the heap. This command can fault if the evaluation of the expression does, or if the contents are not the address of a heap cell. It is deterministic. Similarly, the cell update command alters the contents of a heap cell referenced by an expression. If either expression faults, or the first expression does not reference a flat heap cell, the command faults. The cell dereference action is similar to normal variable assignment, but looks up the result of evaluating the expression in the heap. We could provide an expression for heap cell dereference, but instead use a command. This will simplify reasoning about commands that use expression evaluation.

2.1.3. Operational semantics

We give the meaning of programs via an operational semantics. Our semantics defines a small-step transition system, relating pairs of programs and structured heaps to triples of action labels, continuation programs and outcomes. The action label indicates what interactions a step may have with the structured heap. These actions will form a trace which, along with the heap at each step, can be seen as the meaning of a program. As in [12], these trace semantics are necessary to provide program reasoning for the parallel composition rule. Moreover, by associating actions with each step taken during evaluation, the soundness of the reasoning will be easier to prove.

Definition 5 (Action labels)**.** Given a set of boolean expression BoolExprs (parameter 3) and atomic commands AtomicCmds (parameter 5), the set of **action labels** ActionLbls, ranged over by $\alpha, \alpha_1, \cdots, \alpha_n$, consists of labels for: the identity action; the action of evaluation for each boolean expression to either true, false or fault; and the action of each atomic command:

$$
\begin{array}{lll}
 & & \textbf{Label denotes...} \\
\textsc{ActionLbls} \triangleq & \{\textsc{id}\} & \text{no interaction with heap} \\
\cup & \{\mathcal{E}_{\top}(\textsc{b}) \mid \textsc{b} \in \textsc{BoolExprs}\} & \text{evaluation of b on heap, resulting in true} \\
\cup & \{\mathcal{E}_{\bot}(\textsc{b}) \mid \textsc{b} \in \textsc{BoolExprs}\} & \text{evaluation of b on heap, resulting in false} \\
\cup & \{\mathcal{E}_{\lightning}(\textsc{b}) \mid \textsc{b} \in \textsc{BoolExprs}\} & \text{evaluation of b on heap, resulting in fault} \\
\cup & \{\mathcal{A}(\textsc{c}) \mid \textsc{c} \in \textsc{AtomicCmds}\} & \text{execution of atomic command c on heap}
\end{array}
$$

The use of different labels for the outcome of evaluating boolean expressions will ease the proof when showing soundness of the `if` and `while` rules in our reasoning system.

With these labels, we can define the single-step transition relation for our operational semantics. It represents one "computation step" of an imperative machine.

Definition 6 (Single-step labelled transition relation)**.** Given programs Programs (definition 4), structured heaps StructHeaps (parameter 2), atomic command actions $[\![\text{C}]\!](\cdot)$ (parameter 6), action labels ActionLbls (definition 5), and outcome states Outcomes (definition 2), the **labelled single-step transition relation** $\longrightarrow \subset (\text{Programs} \times \text{StructHeaps}) \times (\text{ActionLbls} \times \text{Programs} \times \text{Outcomes})$, with elements $((\mathbb{C}_1, s_1), (\alpha, \mathbb{C}_2, s_2)) \in \longrightarrow$ written $\mathbb{C}_1, s_1 \xrightarrow{\alpha} \mathbb{C}_2, s_2$, is defined by the following rules:

Atomic command:

$$\frac{s_o \in [\![\text{C}]\!](s)}{\text{C}, s \xrightarrow{\mathcal{A}(\text{C})} \texttt{skip}, s_o}$$

Seq skip:

$$\frac{}{(\texttt{skip};\ \mathbb{C}), s \xrightarrow{\text{ID}} \mathbb{C}, s}$$

Seq reduce:

$$\frac{\mathbb{C}_1, s \xrightarrow{\alpha} \mathbb{C}_1', s_o}{(\mathbb{C}_1;\ \mathbb{C}_2), s \xrightarrow{\alpha} (\mathbb{C}_1';\ \mathbb{C}_2), s_o}$$

If true:

$$\frac{[\![\text{B}]\!](s) = \textbf{true}}{\texttt{if (B) } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, s \xrightarrow{\mathcal{E}_\top(\text{B})} \mathbb{C}_1, s}$$

If false:

$$\frac{[\![\text{B}]\!](s) = \textbf{false}}{\texttt{if (B) } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, s \xrightarrow{\mathcal{E}_\bot(\text{B})} \mathbb{C}_2, s}$$

If fault:

$$\frac{[\![\text{B}]\!](s) = \lightning}{\texttt{if (B) } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, s \xrightarrow{\mathcal{E}_\lightning(\text{B})} \texttt{skip}, \lightning}$$

While true:

$$\frac{[\![\text{B}]\!](s) = \textbf{true}}{\texttt{while (B) } \mathbb{C}_1, s \xrightarrow{\mathcal{E}_\top(\text{B})} (\mathbb{C}_1; \texttt{ while (B) } \mathbb{C}_1), s}$$

While false:

$$\frac{[\![\text{B}]\!](s) = \textbf{false}}{\texttt{while (B) } \mathbb{C}_1, s \xrightarrow{\mathcal{E}_\bot(\text{B})} \texttt{skip}, s}$$

While fault:

$$\frac{[\![\text{B}]\!](s) = \lightning}{\texttt{while (B) } \mathbb{C}_1, s \xrightarrow{\mathcal{E}_\lightning(\text{B})} \texttt{skip}, \lightning}$$

PAR LEFT REDUCE:

$$\frac{\mathbb{C}_1, s \xrightarrow{\alpha} \mathbb{C}_1', s_o}{(\mathbb{C}_1 \mid\mid \mathbb{C}_2) \xrightarrow{\alpha} (\mathbb{C}_1' \mid\mid \mathbb{C}_2), s_o}$$

PAR RIGHT REDUCE:

$$\frac{\mathbb{C}_2, s \xrightarrow{\alpha} \mathbb{C}_2', s_o}{(\mathbb{C}_1 \mid\mid \mathbb{C}_2) \xrightarrow{\alpha} (\mathbb{C}_1 \mid\mid \mathbb{C}_2'), s_o}$$

PAR LEFT SKIP:

$$\frac{}{(\texttt{skip} \mid\mid \mathbb{C}), s \xrightarrow{\text{ID}} \mathbb{C}, s}$$

PAR RIGHT SKIP:

$$\frac{}{(\mathbb{C} \mid\mid \texttt{skip}), s \xrightarrow{\text{ID}} \mathbb{C}, s}$$

There is no transition for the program `skip`. This program is considered to be completely executed, and to have **terminated**.

Via the small-step transition relation, we define a multi-step program evaluation relation which takes programs through many small-steps to a final outcome. It is effectively the reflexive, transitive closure of the single-step evaluation relation, but checks that the outcome of each step is not fault before taking another. If any step does produce fault, the multi-step evaluation relation immediately relates the program to fault.

Definition 7 (Multi-step program evaluation relation)**.** Given programs PROGRAMS (definition 4), structured heaps STRUCTHEAPS (parameter 2) outcome states OUTCOMES (definition 2), and the labelled single-step transition relation (definition 6), the **multi-step program evaluation relation** $\rightsquigarrow \subset (\text{PROGRAMS} \times \text{STRUCTHEAPS}) \times \text{OUTCOMES}$, with elements $((\mathbb{C}, s), s_o) \in \rightsquigarrow$ written $\mathbb{C}, s \rightsquigarrow s_o$, is defined by the following rules:

$$\frac{\mathbb{C}, s \xrightarrow{\alpha} \mathbb{C}', s_o \quad s_o \in \text{STRUCTHEAPS} \quad \mathbb{C}', s_o \rightsquigarrow {s_o}'}{\mathbb{C}, s \rightsquigarrow {s_o}'}$$

$$\frac{}{\texttt{skip}, s \rightsquigarrow s} \qquad \frac{\mathbb{C}, s \xrightarrow{\alpha} \mathbb{C}', \lightning}{\mathbb{C}, s \rightsquigarrow \lightning}$$

Notice not all programs will have an outcome in the multi-step evaluation; specifically, programs with infinite loops never reduce to an outcome. However, programs that fault always have an outcome, as they immediately reduce to the fault state. Programs therefore do one of three things: *diverge*, never terminating nor faulting; *terminate*, ending with some structured heap; or *fault*, failing somehow. We call the process of a machine applying the transition rules *program execution* (or *running* a program).

By examining the transition labels on each step used in the multi-step evaluation, we can see a program execution as a trace of interactions with the heap. Every label indicates either no interaction (the ID label), an interaction that does not update the heap (evaluating a boolean expression, the $\mathcal{E}$ labels), or arbitrary update (performing an atomic command, $\mathcal{A}(\text{C})$). As programs may not terminate, traces need not be finite. Each program can be associated with many traces due to command non-determinism.

Example 7 (Traces)**.** The simple program `x := 1; y := 2` has the trace $\mathcal{A}(\texttt{x := 1}), \mathcal{A}(\texttt{y := 2})$, indicating the two actions on the heap are performed in order. The program `while (true) skip` has the infinite trace $\mathcal{E}_\top(\texttt{true}), \text{ID}, \mathcal{E}_\top(\texttt{true}), \text{ID}, \mathcal{E}_\top(\texttt{true}), \text{ID}, \cdots$. By examining the literal expression `true`, it is evident no heap access is performed. The program `x := 1 || y := 2` has the two possible traces $\mathcal{A}(\texttt{x := 1}), \mathcal{A}(\texttt{y := 2})$ and $\mathcal{A}(\texttt{y := 2}), \mathcal{A}(\texttt{x := 1})$, because parallel composition interleaves evaluation in a non-deterministic order.

2.2. Abstracting program states

We now turn to reasoning about imperative machines, providing *local* reasoning in the style of separation logic. To give pre- and post-conditions for programs, we first build an abstract representation of structured heaps. These are sets of *instrumented structured heaps*. The types of instrumentation depend on the goals of the reasoning being performed. Many separation logics (e.g. [63]) require no instrumentation. However, concurrent separation logic benefits from *fractional permissions* - rational numbers associated with values indicating ownership. In chapter 3, we will use rich instrumentation to enable us to split abstract structured data.

2.2.1. Instrumented structured heaps

Structured heaps provide the state of imperative machines. We now define *instrumented structured heaps*. Sets of these can informally be seen as the "states" of the *program reasoning*, and will be used to abstract the possible states of an imperative machine. These sets will provide the pre- and post-condition triples of our Hoare logic. A single instrumented structured heap consists of instrumented addresses mapping to instrumented values.

Parameter 7 (Instrumented addresses and values)**.** Assume a set of **instrumented addresses** InstAddrs, ranged over by $\mathbf{a}, \mathbf{a}_1, \cdots, \mathbf{a}_n$. Assume also a set of **instrumented values** InstVals, ranged over by $\mathbf{v}, \mathbf{v}_1, \cdots, \mathbf{v}_n$.

Parameter 8 (Instrumented structured heaps)**.** Given sets of instrumented addresses InstAddrs and values InstVals (parameter 7), assume a set of **instrumented structured heaps** InstHeaps, ranged over by $\mathbf{s}, \mathbf{s}_1, \cdots, \mathbf{s}_n$, with type:

$$\textsc{InstHeaps} \subseteq \{\mathbf{s} \mid \mathbf{s} : \textsc{InstAddrs} \xrightarrow{\text{fin}} \textsc{InstVals}\}$$

Define also the **empty instrumented structured heap $\mathbf{0_s}$** as the function with empty domain and co-domain, $\mathbf{0_s} \triangleq \{\} \mapsto \{\}$.

Example 8 (Instrumented primitive heaps)**.** Primitive heaps (example 4) can be used without any instrumentation, as in the original separation logic work [52]. The structured heaps used for the reasoning are therefore identical to the structured heaps used by the machine. Let $\textsc{InstAddrs} = \textsc{PVars} \cup \mathbb{N}^+$, and $\textsc{InstVals} = \textsc{PVals}$, and define the instrumented structured heaps as:

$$\textsc{InstHeaps} \triangleq \left\{ s \;\middle|\; \begin{array}{l} s : \textsc{PVars} \cup \mathbb{N}^+ \xrightarrow{\text{fin}} \textsc{PVals}, \\ \forall a \in \text{dom}(s).\ a \in \mathbb{N}^+ \implies s(a) \in \mathbb{N} \end{array} \right\}$$

Example 9 (Primitive heaps with fractional permissions)**.** Recall the fractional permissions introduced in section 1.2.1. We can create instrumented structured heaps that use these permissions by pairing values with some $\pi \in (0, 1]$. Let $\textsc{InstAddrs} = \textsc{PVars} \cup \mathbb{N}^+$ as in example 8, but now define $\textsc{InstVals} = \textsc{PVals} \times \mathbb{Q}$. The instrumented structured heaps become:

$$\textsc{InstHeaps} \triangleq \left\{ s \;\middle|\; \begin{array}{l} s : \textsc{InstAddrs} \xrightarrow{\text{fin}} \textsc{InstVals}, \\ \forall a \in \text{dom}(s).\ a \in \mathbb{N}^+ \implies (s(a)\downarrow_1 \in \mathbb{N} \wedge s(a)\downarrow_2 \in (0, 1]) \end{array} \right\}$$

This definition mirrors that of the heaps for normal primitive heaps but checks that each fraction associated with the flat heap cells is in the range $(0, 1]$.

An instrumented heap can be *reified* into a set of structured heaps. Reification gives an interpretation of the instrumented heap, typically based upon the instrumentation it is using. There are no constraints a reification function. The choice endows a *meaning* or *interpretation* to instrumented heaps, which will later define the kinds of properties we can prove. We first define *primitive reification*, which works on a single instrumented heap. Full reification will be defined on sets of heaps as the pointwise lift of this function.

Parameter 9 (Primitive reification function)**.** Given instrumented heaps InstHeaps (parameter 8) and structured heaps StructHeaps (parameter 2), assume a **primitive reification function** $[\![\cdot]\!] : \textsc{InstHeaps} \to \mathcal{P}(\textsc{StructHeaps})$.

Notice that primitive reification does not generate outcome states, only structured heaps. Thus, the reification of an instrumented heap never includes the fault state. This *fault avoiding* reification ensures that the machine states described by instrumented heaps are always useful.

The behaviour of primitive reification is dependent on both the instrumentation associated with heaps, and what the user of this framework wishes the abstraction to mean. Sometimes, as in the primitive heap case (example 4), we need no instrumentation. In this case, reification will be simple.

Example 10 (Reification for primitive heaps)**.** The instrumented primitive heaps of example 8 have no instrumentation. Pick the structured heaps as those of 4, and then select the primitive reification function as $[\![\mathbf{s}]\!] = \{\mathbf{s}\}$.

As the reification function is a free choice, we can create quite different interpretations of the same underlying instrumented heaps.

Example 11 (Alternative reification for primitive heaps)**.** An alternative choice of primitive reification for primitive heaps interprets the instrumented heap as every primitive structured heap which is *consistent* with the data in the instrumented heap:

$$[\![\mathbf{s}]\!] = \{s \in \textsc{StructHeaps} \mid \forall a \in \mathrm{dom}(\mathbf{s}).\ s(a) = \mathbf{s}(a)\}$$

Here, the instrumented heap $\{1 \mapsto 5\}$ primitively reifies to *every* heap in which 1 addresses value 5. We will see how the choice of primitive reification in example 10 generates program reasoning similar to *classical* separation logic. This alternative choice generates reasoning similar to *intuitionistic* separation logic. For the rest of this chapter, we will work with the first choice unless otherwise stated.

Often, instrumentation must be removed to generate machine heaps, as in the case of the fractional permissions (example 9).

Example 12 (Reification for primitive heaps with fractional permissions)**.** Fractional permissions are used purely to facilitate reasoning correctness. They have no bearing on the underlying structured heaps, so we pick a primitive reification function that just erases the permission:

$$\llbracket \mathbf{s} \rrbracket = \lambda a.\ \mathbf{s}(a)\!\downarrow_1$$

Instrumented heaps can describe *partial* machine states. For example, the instrumented heap $\mathtt{x} \overset{0.5}{\to} 1$ (example 9) describes only half permission on a heap cell, implying that the other half is missing. It is thus natural to compose instrumented heaps. To this end, we introduce *primitive instrumented heap composition*, allowing two instrumented heaps to be joined. Full composition will be the pointwise lift of this to sets of instrumented heaps.

Parameter 10 (Primitive instrumented heap composition)**.** Given instrumented heaps InstHeaps with their unit $\mathbf{0_s}$ (parameter 8), assume a **primitive instrumented heap composition** operator:

$$\circledast : \textsc{InstHeaps} \to \textsc{InstHeaps} \rightharpoonup \textsc{InstHeaps}$$

where the operator is associative and commutative with unit $\mathbf{0_s}$. That is, for all $\mathbf{s}, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3 \in \textsc{InstHeaps}$ $(\mathbf{s}_1 \circledast \mathbf{s}_2) \circledast \mathbf{s}_3 = \mathbf{s}_1 \circledast (\mathbf{s}_2 \circledast \mathbf{s}_3)$, $\mathbf{s}_1 \circledast \mathbf{s}_2 = \mathbf{s}_2 \circledast \mathbf{s}_1$ and $\mathbf{s} \circledast \mathbf{0_s} = \mathbf{s}$.

The choice of primitive composition function depends on the instrumentation being used. Indeed, a key use of instrumentation is to restrict the set of valid compositions. In the separation logic example (example 17), a good choice is disjoint function union, allowing two heaps to be composed only if they share no addresses in common. However,

when using fractional permissions, two heaps can compose even if they have addresses in common, as long as the sum of the permissions associated with these is at most 1. Good choices of composition are those that enable flexible *decomposition*, as composition implicitly defines *separation* ($\mathbf{s} = \mathbf{s}_1 \circledast \mathbf{s}_2$ can be read both as "$\mathbf{s}_1 \circledast \mathbf{s}_2$ compose to give $\mathbf{s}$", but also as "$\mathbf{s}$ decomposes to give $\mathbf{s}_1 \circledast \mathbf{s}_2$, where $\mathbf{s}_1$ and $\mathbf{s}_2$ are separate").

Example 13 (Primitive composition for primitive heaps)**.** For instrumented primitive heaps (example 8), primitive composition joins heaps with disjoint domains (that is, that share no variables or heap cells):

$$\mathbf{s}_1 \circledast \mathbf{s}_2 \triangleq \begin{cases} \mathbf{s}_1 \cup \mathbf{s}_2 & \text{if } \mathrm{dom}(\mathbf{s}_1) \cap \mathrm{dom}(\mathbf{s}_2) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example 14 (Primitive composition with fractional permissions)**.** For primitive heaps using fractional permissions (example 9), primitive composition joins instrumented states by taking the function union, but in the case of an address being in both, *adding* the permission. If any permission would exceed 1, composition is undefined.

$$\mathbf{s}_1 \circledast \mathbf{s}_2 \triangleq \begin{cases} \text{undefined if } \exists \mathbf{a} \in \mathrm{dom}(\mathbf{s}_1) \cap \mathrm{dom}(\mathbf{s}_2) \text{ s.t. } \mathbf{s}_1(\mathbf{a})\downarrow_2 + \mathbf{s}_2(\mathbf{a})\downarrow_2 > 1 \\ \lambda \mathbf{a}. \begin{cases} \mathbf{s}_1(\mathbf{a}) & \text{if } \mathbf{a} \in \mathrm{dom}(\mathbf{s}_1) \setminus \mathrm{dom}(\mathbf{s}_2) \\ \mathbf{s}_2(\mathbf{a}) & \text{if } \mathbf{a} \in \mathrm{dom}(\mathbf{s}_2) \setminus \mathrm{dom}(\mathbf{s}_1) \\ (\mathbf{s}_1(\mathbf{a})\downarrow_1, \mathbf{s}_1(\mathbf{a})\downarrow_2 + \mathbf{s}_2(\mathbf{a})\downarrow_2) & \text{if } \mathbf{s}_1(\mathbf{a})\downarrow_1 = \mathbf{s}_2(\mathbf{a})\downarrow_1 \\ \text{undefined} & \text{otherwise} \end{cases} \end{cases}$$

Notice that the addition of fractional permissions has allowed *more separation* than is possible with composition for primitive heaps (example 13).

2.2.2. Views

We are now in a position to define *views* over instrumented structured heaps. The views system, introduced in [22] with a summary given in section 1.2.2, is a program reasoning

framework based around state abstractions also known as views. Our views are formed from choices of instrumented heaps, with the associated primitive reification and composition functions. We call this collection of objects an *addressed value view*.

Definition 8 (Addressed value view)**.** Given a set of structured heaps StructHeaps and instrumented structured heaps for them InstHeaps (parameter 2 and parameter 8 respectively), a primitive reification function $[\![\cdot]\!] : \textsc{InstHeaps} \to \mathcal{P}(\textsc{StructHeaps})$ (parameter 9) and a primitive composition function with unit $\mathbf{0_s}$, $\circledast : \textsc{InstHeaps} \to \textsc{InstHeaps} \rightharpoonup \textsc{InstHeaps}$ (parameter 10), an **addressed value view** consists of a set of **views** Views, a **reification function** $\lfloor\cdot\rfloor$, a binary **composition operator** $*$, and a **unit element** 0:

$$\text{Addressed value view} = \left(\begin{array}{c} \textsc{Views}, \lfloor\cdot\rfloor : \textsc{Views} \to \mathcal{P}(\textsc{StructHeaps}), \\ * : \textsc{Views} \to \textsc{Views} \rightharpoonup \textsc{Views}, 0 \end{array} \right)$$

with the properties:

1. $\textsc{Views} \triangleq \mathcal{P}(\textsc{InstHeaps})$ is a set of **views**, ranged over by $p, p_1, \cdots, p_n, p, q, r$.
2. The **reification function**, $\lfloor\cdot\rfloor : \textsc{Views} \to \mathcal{P}(\textsc{StructHeaps})$ is defined as the pointwise lift of primitive reification:

$$\lfloor p \rfloor \triangleq \bigcup \{ [\![\mathbf{s}]\!] \mid \mathbf{s} \in p \}$$

3. The **composition function** $* : \textsc{Views} \to \textsc{Views} \rightharpoonup \textsc{Views}$ is the pointwise lift of primitive composition:

$$p * q \triangleq \{ \mathbf{s}_1 \circledast \mathbf{s}_2 \mid \mathbf{s}_1 \in p, \mathbf{s}_2 \in q \}$$

4. 0 is a unit of $*$, defined as $0 \triangleq \{\mathbf{0_s}\}$.

Addressed value views are views in the sense of [22], and are similar to those the authors constructed from separation algebras. However, we allow a choice of primitive reification function in our construction rather than always using the identity, and use our instrumented heaps rather than arbitrary cancellative monoids for constructing the views.

Each view is a "perspective" on the possible underlying machine states. A single view does not tell us which machine state we will be in, and typically provides a conservative

approximation. The compositional nature of views means that these perspectives can be joined, generally resulting in richer knowledge of, and capabilities over, the underlying states. If two views are *incompatible*, then we have conflicting facts about the machine states, and thus no useful information at all.

Comment 2. I find it helpful to anthropomorphise views. Consider a view as an entity that holds a set of opinions about the world. The opinions are the instrumented states, and the world is the set of machine states that reify from them. Composition attempts to form an agreement between two views about the actual states of the world, with the resultant entity having somehow merged the knowledge of both. In this sense, $*$ is forming a *consensus* between parties about what the machine is actually doing. When the parties either hold opinions about different parts of the world (i.e. separation logic), or have the same opinions about the same part (i.e. separation logic with fractional permissions), this is easy. When the parties have differing opinions about the same parts of the world, forming a consensus can be harder, corresponding to more complex logics such as Rely-Guarantee [47] or CAP [23].

We define a set of useful functions and relations over addressed value views.

Definition 9 (Operations on views)**.** Given an addressed value view $(\textsc{Views}, [\![\cdot]\!], \circledast, 0)$ (definition 8), and recalling that Views is formed of sets of instrumented structured heaps InstHeaps (parameter 8), we define notions of entailment, disjunction, conjunction and negation on the set Views as:

1. **Entailment**: The **entailment relation** on views, $\models \subset \textsc{Views} \times \textsc{Views}$, is the subset relation: $\models \triangleq \subseteq$.
2. **Disjunction**: The **disjunction** function on views, $\bigvee : \mathcal{P}(\textsc{Views}) \to \textsc{Views}$, is standard set union: $\bigvee \triangleq \bigcup$.
3. **Conjunction**: The **conjunction** function on views, $\bigwedge : \mathcal{P}(\textsc{Views}) \to \textsc{Views}$, is standard set intersection: $\bigwedge \triangleq \bigcap$.
4. **Negation**: The **negation** function on views, $\neg : \textsc{Views} \to \textsc{Views}$, is the standard set complement: $\neg p \triangleq \textsc{InstHeaps} \setminus p$.

One can informally consider views as "logical formulæ" with instrumented states as their "models". For example, the first operation (entailment) ensures that every instrumented state in the left hand side is contained within the right hand side. This reflects the

operation of logical entailment, in that it holds if all "models" of the left view are "models" of the right. Disjunction gives a view which contains *every* "model" of the parameters. The other two have similar readings.

Comment 3. Whilst this informal notion of views as logical formulæ with instrumented states as models can aid understanding, it is important to note it is *not* a logic in any standard sense. Critically, I provide no syntactic proof rules that deduce truths about views (only rules that deduce truths about programs). Their manipulation remains purely "semantic", in that one must consider the contents and reifications of their component instrumented heaps.

Two useful sets of views are the *boolean expression truth views*. These views reify to the structured heaps in which a given boolean expression will evaluate to true (or false). These views will be used to define the pre- and post-conditions of the `if` and `while` rules of the program logic (theorem 1), which will require analysis of boolean guard expressions.

Definition 10 (Boolean truth views)**.** Let $\mathrm{B} \in \textsc{BoolExprs}$ be any boolean expression (parameter 3) with associated boolean expression evaluation function $[\![\cdot]\!](\cdot)$ (parameter 4). The **truth views** of B are defined as:

$$\mathrm{exprTrue}(\mathrm{B}) \triangleq \{\mathbf{s} \in \textsc{InstHeaps} \mid \forall s \in \lfloor\!\lfloor \mathbf{s} \rfloor\!\rfloor.\ [\![\mathrm{B}]\!](s) = \mathbf{true}\}$$
$$\mathrm{exprFalse}(\mathrm{B}) \triangleq \{\mathbf{s} \in \textsc{InstHeaps} \mid \forall s \in \lfloor\!\lfloor \mathbf{s} \rfloor\!\rfloor.\ [\![\mathrm{B}]\!](s) = \mathbf{false}\}$$

Boolean expression evaluation (parameter 4) is deterministic, and results in one of true, false or fault. Therefore, if an expression evaluates to either true or false, it *cannot* evaluate to fault. We define the *boolean safety view* as the disjunction of the two truth views, and so describe the structured heaps in which a boolean expression can never evaluate to fault.

Definition 11 (Boolean safety view)**.** Let $\mathrm{B} \in \textsc{BoolExprs}$ be any boolean expression (parameter 3), and $\vee$ be disjunction on views (definition 9). The associated **boolean expression safety view** is defined as:

$$\mathrm{safe}(\mathrm{B}) \triangleq \mathrm{exprTrue}(\mathrm{B}) \vee \mathrm{exprFalse}(\mathrm{B})$$

Example 15 (Primitive heaps expression safety)**.** In addition to boolean expressions, our primitive heaps example also has basic expressions (example 5). We can define an expression safety view for these. Let $e \in \textsc{Exprs}$ be any basic expression. The associated **basic expression safety view** is defined as:

$$\text{safe}(e) \triangleq \{\mathbf{s} \in \textsc{InstHeaps} \mid \forall s \in \lfloor\!\lfloor \mathbf{s} \rfloor\!\rfloor.\ (\!|e|\!)(s) \neq \lightning\}$$

2.3. Program reasoning

We now introduce our program reasoning framework, utilising views as the pre- and post-conditions for programs.

2.3.1. Semantic Hoare triples

We call a triple consisting of a program with associated pre- and post-condition views a *semantic Hoare triple*.

Definition 12 (Semantic Hoare triples)**.** Given an addressed value view $(\textsc{Views}, \lfloor\cdot\rfloor, *, 0)$ (definition 8), and programs $\textsc{Programs}$ (definition 4), the set of **semantic Hoare triples** $\textsc{ViewTriples}$, ranged over by $vt, vt_1, \cdots, vt_n$, is defined as:

$$\textsc{ViewTriples} \triangleq \textsc{Views} \times \textsc{Programs} \times \textsc{Views}$$

An element $(p, \mathbb{C}, q) \in \textsc{ViewTriples}$ is written $\{p\}\ \mathbb{C}\ \{q\}$.

We use the term *semantic* because the pre-and post-conditions are views, rather than syntactic assertions. The intended meaning of a semantic Hoare triple $\{p\}\ \mathbb{C}\ \{q\}$ is that, when $\mathbb{C}$ is run with a heap contained within the reification of view p, then it will never fault and, if the program terminates, the outcome will be a heap reified from view q.

To formalise this interpretation, first recall the action labels of definition 5. Each label represents the interaction of a program step with a structured heap: either no interaction; the evaluation of an expression; or the application of an atomic command. In a similar vein, we can define the interaction of a label with a pair of views p, q representing the program state before and after the action associated with the label. This *action judgement* considers the effect of the action on the structured heaps generated by reifying p and q.

More than this, it considers the effects when those views are composed with an arbitrary frame $r \in \textsc{Views}$.

We do this as we are developing a local Hoare reasoning framework. Commands are specified using views that describe *only* the resource they need. They are free to update the resource in their pre- and post-conditions as needed, but must *not* disturb resource their specification does not mention. By "baking" all possible frames r into the judgement, unchanged on both sides, we are assured that unmentioned resource is not meaningfully changed, a property called *stability*.

Definition 13 (Action judgement)**.** Given a set of action labels (definition 5) and an addressed value view ($\textsc{Views}, \lfloor\cdot\rfloor, *, 0$) (definition 8), the **action judgement** $\cdot \models \{\cdot\}\{\cdot\} \subseteq \textsc{ActionLbls} \times \textsc{Views} \times \textsc{Views}$ is the largest relation such that:

$$\alpha \models \{p\}\{q\} \iff \forall r \in \textsc{Views}.\ [\alpha](\lfloor p * r \rfloor) \subseteq \lfloor q * r \rfloor$$

where, given that $[\![\textsc{c}]\!]$ is the action of atomic command $\textsc{c}$ (parameter 6), the function $[\cdot]\,(\cdot) : \textsc{ActionLbls} \to \mathcal{P}(\textsc{StructHeaps}) \to \mathcal{P}(\textsc{StructHeaps})$ is defined as:

$$\begin{aligned}
[\textsc{id}]\,(S) &\triangleq S \\
[\mathcal{E}_\top(\textsc{b})]\,(S) &\triangleq \{s \in S \mid [\![\textsc{b}]\!](s) = \mathbf{true}\} \\
[\mathcal{E}_\bot(\textsc{b})]\,(S) &\triangleq \{s \in S \mid [\![\textsc{b}]\!](s) = \mathbf{false}\} \\
[\mathcal{E}_{\lightning}(\textsc{b})]\,(S) &\triangleq \{s \in S \mid [\![\textsc{b}]\!](s) = \lightning\} \\
[\mathcal{A}(\textsc{c})]\,(S) &\triangleq \bigcup\{[\![\textsc{c}]\!](s) \mid s \in S\}
\end{aligned}$$

Informally, the action judgement $\alpha \models \{p\}\{q\}$ says that applying action α to the states described by p results in states described by q. Notice that the action judgement associated with boolean expression evaluation requires that the post-condition have restricted the set of heaps to those that result in the outcome indicated on the label. These actions are used to direct the flow control in cases of `if` and `while`, and this filtering of heaps matches the operational effect of the label. For example, operationally, the first branch of `if` retains only those states which evaluate the guard expression to true. By having the action judgement perform the same filtering, soundness of the reasoning is easier to prove.

The "baking in" of the frame ensures the *locality* of the action judgement, and the stability of resource not mentioned. Locality is a key property. Assume that $\alpha \models \{p\}\{q\}$. Then, by definition, both of the follow properties hold.

1. **Extensibility:** We can compose p and q with any arbitrary view r, and the judge-

ment will still hold. This property allows knowledge about interactions with "small" heaps to be extended to knowledge about larger heaps, which is the essence of the frame rule of local reasoning.

2. **Locality:** The action labelled α is *local* to the resource described by p and q. These resources must be all it needed, and all it can have meaningfully changed. To see p is all that is needed, imagine the action needed more resource than can be reified from p. By picking $r = 0$, the command would then fault on heaps from $\lfloor p \rfloor$. However, fault is not contained in any reification (parameter 9), so cannot be within $\lfloor q \rfloor$, and so the judgement could not have held. To see that nothing else is altered, pick arbitrary r. For all non-divergent commands, it must be the case that $q * r$ is also defined. If the action had altered resource outside of p, we would then be able to find some r that represented this resource, yet must be unchanged when composed with q. Thus, the types of changes the action could perform on resource not represented by p must be trivial, and so r must be stable.

Via the action judgement, we formalise the meaning of a semantic triple (definition 12).

Definition 14 (Semantic triple judgement)**.** Given structured heaps StructHeaps (parameter 2) and action labels ActionLbls (definition 5), for all programs $\mathbb{C}_1 \in$ Programs (definition 4) and views $p_1, q \in$ Views (definition 8), the **semantic triple judgement** $\vDash \{\cdot\} \ \cdot \ \{\cdot\} \subseteq$ ViewTriples is defined as the largest relation such that $\vDash \{p_1\}\ \mathbb{C}_1\ \{q\}$ if and only if:

1. if $\mathbb{C}_1 = \texttt{skip}$, then the action judgement $\text{ID} \Vdash \{p_1\}\{q\}$ holds.
2. if $\mathbb{C}_1 \neq \texttt{skip}$, then for all action labels $\alpha \in$ ActionLbls and structured heaps $s_1 \in \lfloor p_1 \rfloor$, if there is some $s_2 \in$ StructHeaps and $\mathbb{C}_2 \in$ Programs such that $\mathbb{C}_1, s_1 \xrightarrow{\alpha} \mathbb{C}_2, s_2$, then there exists some $p_2 \in$ Views such that:
   a) $\alpha \Vdash \{p_1\}\{p_2\}$
   b) $\vDash \{p_2\}\ \mathbb{C}_2\ \{q\}$

Consider figure 2.2, which gives an example program $\mathbb{C}$ with associated views, traces and view reifications. The views column represents a list of views generated by unfolding the semantic triple judgement $\vDash \{p_1\}\ \mathbb{C}\ \{q\}$. The judgement forces the views to be a good *abstraction* of the possible structured heaps at each state, no matter what program trace actually occurs. Notice that the reified views always contain the possible trace states at each step.

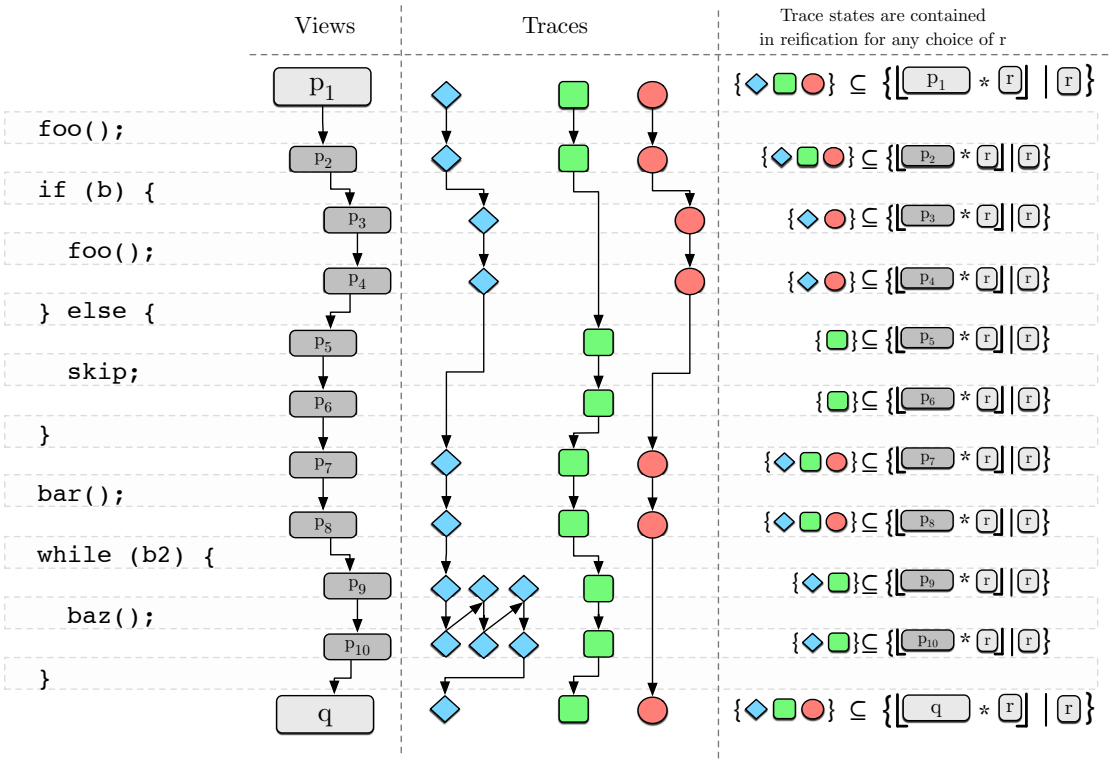


Figure 2.2.: Demonstration of the relationship between a program, views, the possible program action traces, and reified views. This figure represents the unfolding of the semantic triple judgement $\vDash \{p_1\}\ \mathbb{C}\ \{q\}$ (definition 14) and the relationship between the structured heaps reified from the views and possible traces of the program.

2.3.2. Deriving valid semantic Hoare triples

It is prohibitively complex to construct valid semantic Hoare triples by analysing an entire program. Instead, we provide a set of *semantic inferences* that allow the construction of correct triples. We start with the program primitives; i.e. those programs not composed from smaller programs.

Proposition 1 (Skip judgement)**.** For any view $p \in \textsc{Views}$, $\vDash \{p\}\ \texttt{skip}\ \{p\}$ is a valid semantic triple judgement.

Proof. As $\mathbb{C} = \texttt{skip}$, the judgement falls into the first case of definition 14. Hence, we must show $\textsc{id} \Vdash \{p\}\{p\}$. This holds trivially, as $\forall r.\ \lfloor p * r \rfloor \subseteq \lfloor p * r \rfloor$ always. □

Atomic commands are given semantic triple specifications via axioms. All atomic commands must have at least one axiom.

Parameter 11 (Atomic command axioms)**.** Given a set of atomic commands $\textsc{AtomicCmds}$ (parameter 5) and an addressed value view $(\textsc{Views}, \lfloor\cdot\rfloor, *, 0)$ (definition 8), assume a set of **atomic command axioms** $\textsc{Axioms} \subseteq (\textsc{Views} \times \textsc{AtomicCmds} \times \textsc{Views})$ such that for each $\mathrm{c} \in \textsc{AtomicCmds}$, there exists $p, q \in \textsc{Views}$ with $(p, \mathrm{c}, q) \in \textsc{Axioms}$.

These axioms must sound, in that they are valid abstractions of the command actions. This *atomic soundness* property must be proven for each atomic command.

Parameter 12 (Atomic soundness)**.** Given a set of atomic commands $\textsc{AtomicCmds}$ (parameter 5) and the action judgement (definition 13), the **atomic soundness property** of axioms states that, for every $(p, \mathrm{c}, q) \in \textsc{Axioms}$, $\mathcal{A}(\mathrm{c}) \Vdash \{p\}\{q\}$ holds.

Example 16 (Primitive heap axiomatisation)**.** We now give axioms for the commands of primitive heaps (example 6). These commands involve expressions that can access variables in the heap. When the specific expression is unknown, the number of variables involved in the command action is equally unknown. The axioms for these commands must express sufficient additional resource to safely evaluate the expression. To accomplish this, we add additional resource to the specifications of such commands and use the expression safety view of definition 11 to ensure this resource

contains all the needed variables are present.

With an expression and a view containing sufficient variable resource to ensure non-faulting evaluation of the expression, we can describe a view that describes when the expression evaluates to a certain value. Formally, given a set of expressions Exprs evaluated by $(\!|\cdot|\!)(\cdot)$ (definition 5) into program values PVals, and an addressed value view (Views, $\lfloor\cdot\rfloor, *, 0$), the **expression evaluation view** $\Rightarrow : \text{Exprs} \to \text{PVals} \to \text{Views}$, written $e \Rightarrow i$, is defined as

$$e \Rightarrow v \triangleq \{s \mid \lfloor\!|s|\!\rfloor = heaps, \forall h \in heaps.\ (\!|e|\!)(h) = v\}$$

Notice that the view $e \Rightarrow v$ always entails the safety view, $(e \Rightarrow v) \models \text{safe}(e)$ by definition. By using conjunction between some view p and the expression evaluation view, e.g. $p \wedge e \Rightarrow v$, we describe a view which has the variables of p *and* safely evaluates e to v.

Let $\mathtt{x} \to v$ be the singleton view with variable $\mathtt{x}$ mapping to value $\mathtt{x}$, and $i \mapsto j$ be the singleton view with flat heap cell at address i mapping to value j. The axioms are then defined as follows: for all $\mathtt{x} \in \text{PVars}, v \in \text{PVals}, e, e_1, e_2 \in \text{Exprs}$ and $o \in \text{Views}$

$$\{(\mathtt{x} \to v * o) \wedge e \Rightarrow i\} \quad \mathtt{x := } e \quad \{\mathtt{x} \to i * o\}$$

$$\{\mathtt{x} \mapsto v\} \quad \mathtt{x := alloc()} \quad \left\{\bigvee_{i \in \mathbb{N}^+, j \in \mathbb{N}} \mathtt{x} \mapsto i * i \mapsto j\right\}$$

$$\{i \mapsto j * (o \wedge e \Rightarrow i)\} \quad \mathtt{free}(e) \quad \{o\}$$

$$\{i \mapsto j * (o \wedge e_1 \Rightarrow i \wedge e_2 \Rightarrow k)\} \quad \mathtt{[}e_1\mathtt{]\ :=}\ e_2 \quad \{i \mapsto k * o\}$$

$$\{i \mapsto j * ((\mathtt{x} \to v * o) \wedge e \Rightarrow i)\} \quad \mathtt{x := [}e\mathtt{]} \quad \{i \mapsto j * \mathtt{x} \to j * o\}$$

The first axiom, variable/expression assignment, is an example of the extra resource needed by expressions. Consider the atomic actions of this command (definition 6). The expression e is evaluated, which by definition can use arbitrary variable resource. The result of the evaluation is placed in heap cell $\mathtt{x}$. No other resource is read or updated by the action, and so is not mentioned.

We mirror this in the axiom. The first conjunct of the pre-condition, $\mathtt{x} \mapsto v * o$,

describes a heap that contains the cell addressed by `x` with some value. It also contains *arbitrary* additional heap cells that are disjoint from `x`, given by o, that describe additional variables needed to evaluate e. The second conjunct, $e \Rightarrow i$, gives *all* views that describe heaps evaluating e to i. By conjoining them, the pre-condition describes exactly heaps with variable `x` and arbitrary other variables, that safely evaluate e to i. As there is an axiom for *every* choice of i, it does not matter what specific value the expression evaluates to.

The post-condition view $\texttt{x} \to i * o$ is similar to the pre-condition, except that the `x` variable cell now contains the result of evaluating e, and has dropped the conjunction. The unchanged nature of the other cells is assured by the presence of o. The second conjunct is unnecessary, as the first conjunction contains all the information needed. As i was chosen for the entire axiom, the correct result is bound to the variable.

To show atomic soundness, we must show that the post-condition captures the effect of the atomic command on the pre-condition, and that the action is still well-described whenever the conditions are extended with arbitrary view r. The post-condition described above is exactly the effect of the atomic action. Preservation of frames is assured as the command does not touch any cells other than `x` or those mentioned in o. If the frame r mentions one of these cells, the pre-condition becomes the empty set, and so the action judgement holds vacuously.

Turn now to allocation, `x:= alloc()`. Recall that the atomic action for this command (definition 6) is given as:

$$[\![\texttt{x := alloc()}]\!](s) \triangleq \{s[\texttt{x} \mapsto i, i \mapsto v] \mid i \in \mathbb{N}^+ \setminus \mathrm{dom}(s), v \in \mathbb{N}\}$$

The action creates a new flat heap cell at any unused positive integer address with any natural number as a value, *and* updates the variable `x` to the address added. It is thus non-deterministic; we do not know the address that will be created, nor the value it will contain. To prove atomic soundness of the axiom, consider the pre-condition. It describes the singleton instrumented heap with variable `x` mapping to any value. The post-condition describes an infinite set of instrumented heaps, covering every combination of a single flat-heap cell address/value alongside `x` mapping to that address. This infinity of possibilities is the standard method for handling the non-local nature of allocation; we do not know *which* other addresses are used, but we do know there will be free addresses available, and that the command will choose one of them.

Formally, we show:

$$\forall r \in \textsc{Views}.\ [\texttt{x := alloc()}]\lfloor \texttt{x} \mapsto v * r \rfloor \subseteq \left\lfloor \bigvee_{i \in \mathbb{N}^+, j \in \mathbb{N}} (\texttt{x} \mapsto i * i \mapsto j) * r \right\rfloor$$

Fix arbitrary $r \in \textsc{Views}$. Then, by reification being the pointwise lift of primitive reification (example 10), and composition being function union (example 13):

$$\lfloor \{\texttt{x} \mapsto v\} * r \rfloor = \{\{\texttt{x} \mapsto v\} \sqcup s_r \mid s_r \in \lfloor r \rfloor, \texttt{x} \notin \mathrm{dom}(s_r)\}$$

Pick an arbitrary member $s_p \in \lfloor \{\texttt{x} \mapsto v\} * r \rfloor$. Then

$$[\![\texttt{x := alloc()}]\!](s_p) = \{s_p[\texttt{x} \mapsto i, i \mapsto v'] \mid i \in \mathbb{N}^+ \setminus \mathrm{dom}(s_p), v' \in \mathbb{N}\}$$

This result set is non-empty, as each s_p is a finite function on an infinite domain. Pick an arbitrary $s_q \in [\![\texttt{x := alloc()}]\!](s_p)$. Note that

$$\left\lfloor \bigvee_{i \in \mathbb{N}^+, j \in \mathbb{N}} (\texttt{x} \mapsto i, i \mapsto j) * r \right\rfloor = \left\{ \{\texttt{x} \mapsto i, i \mapsto j\} \sqcup s_r \;\middle|\; \begin{array}{l} i \in \mathbb{N}^+, j \in \mathbb{N}, \\ s_r \in \lfloor r \rfloor, \\ \texttt{x} \notin \mathrm{dom}(s_r), \\ i \notin \mathrm{dom}(s_r) \cup \{\texttt{x}\} \end{array} \right\}$$

Call this set S. The result follows if s_q is a member of S. Observe that s_p is any heap in which `x` exists alongside arbitrary other cells s_r added by r. The heap s_q is any heap in which `x` points to an arbitrary $i \in \mathbb{N}^+$ not present in s_r, and in which i points to an arbitrary $v \in \mathbb{N}$. The set S is by definition *every* such heap, ergo $s_q \in S$ as required.

The cases for cell deallocation, assignment and dereference are similar.

The skip rule and axioms provide valid semantic judgements for the most primitive programs. To construct judgements about composite programs, we provide a set of *semantic inference rules*. These inferences take simpler valid judgements and program fragments and build more complex judgements.

Theorem 1 (Semantic inference rules)**.** *Given a set of boolean expressions* BoolExprs *(parameter 3), programs* Programs *(definition 4) and addressed value view* (Views, $\lfloor\cdot\rfloor, *, 0$) *(definition 8), the rules in the following two groups of* ***semantic inference rules*** *are valid, in that if the assumed semantic Hoare triple judgements hold then the conclusion*

semantic Hoare triple judgement holds. The rules are: for all $\text{B} \in \textsc{BoolExprs}$*, programs* $\mathbb{C}, \mathbb{C}_1, \mathbb{C}_2 \in \textsc{Programs}$ *and* $p, p_1, p_2, q, r, o \in \textsc{Views}$

Program rules*: These rules are driven by the syntax of the program.*

1. ***Sequencing rule****: If* $\vDash \{p\}\ \mathbb{C}_1\ \{o\}$ *and* $\vDash \{o\}\ \mathbb{C}_2\ \{q\}$*, then* $\vDash \{p\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \{q\}$.
2. ***If rule****: If* $p \models \mathit{safe}(\text{B})$, $\vDash \{p \wedge \mathit{exprTrue}(\text{B})\}\ \mathbb{C}_1\ \{q\}$ *and* $\vDash \{p \wedge \mathit{exprFalse}(\text{B})\}\ \mathbb{C}_2\ \{q\}$*, then* $\vDash \{p\}$ `if (B)` $\mathbb{C}_1$ `else` $\mathbb{C}_2\ \{q\}$.
3. ***While rule****: If* $p \models \mathit{safe}(\text{B})$ *and* $\vDash \{p \wedge \mathit{exprTrue}(\text{B})\}\ \mathbb{C}\ \{p\}$*, then* $\vDash \{p\}$ `while (`b`)` $\mathbb{C}\ \{p \wedge \mathit{exprFalse}(b)\}$
4. ***Parallel Composition rule****: If* $\vDash \{p_1\}\ \mathbb{C}_1\ \{p_2\}$ *and* $\vDash \{p_2\}\ \mathbb{C}_2\ \{q_2\}$*, then* $\vDash \{p_1 * p_2\}\ \mathbb{C}_1 \mid\mid \mathbb{C}_2\ \{q_2 * q_2\}$.

Abstraction rules*: These rules are driven by the structure of the views abstracting the state.*

1. ***Frame rule****: For all* $r \in \textsc{Views}$*, if* $\vDash \{p\}\ \mathbb{C}\ \{q\}$ *then* $\vDash \{p * r\}\ \mathbb{C}\ \{q * r\}$
2. ***Disjunction rule****: Let* I *be some index set. If, for all* $i \in I$, $\vDash \{p_i\}\ \mathbb{C}\ \{q_i\}$*, then* $\vDash \{\bigvee_{i \in I} p_i\}\ \mathbb{C}\ \{\bigvee_{i \in I} q_i\}$.
3. ***Consequence rule****: For all* $p', q' \in \textsc{Views}$*, if* $p \models p'$, $q' \models q$ *and* $\vDash \{p'\}\ \mathbb{C}\ \{q'\}$ *then* $\vDash \{p\}\ \mathbb{C}\ \{q\}$.

Proof. The semantic triple judgement $\vDash \{p_1\}\ \mathbb{C}_1\ \{q\}$ (definition 14) is defined co-inductively. The proof is therefore co-inductive. Unpacking the definition, we must show that:

1. If the program has finished (that is, it is `skip`), the structured heaps reified from the triple pre-condition p_1 are contained within those reified from the post-condition q, when considered under all frames. Formally, $\mathbb{C}_1 =$ `skip` implies $\textsc{id} \Vdash \{p_1\}\{q\}$.
2. If the program has not finished, and so takes a step to a new state, then there is *some* view that both abstracts the structured heap after the step, and functions as a pre-condition for the continuation program. Formally, for all $s_1 \in \lfloor p_1 \rfloor$, if there are some $\alpha, \mathbb{C}_2$ and s_2 such that $\mathbb{C}_1, s_1 \xrightarrow{\alpha} \mathbb{C}_2, s_2$, then is some view p_2 such that $\alpha \Vdash \{p_1\}\{p_2\}$ and $\vDash \{p_2\}\ \mathbb{C}_2\ \{q\}$.

Referring again to figure 2.2, this proof justifies the existence of the sequence of views that abstract the structured heaps. In essence, the uses of co-induction create a simulation relation, demonstrating that for every "step" the assumed judgements take with the action judgement, the conclusion judgement can match the "step".

1. **Sequencing rule**: We proceed by co-induction. Assume ① $\vDash \{p\}\ \mathbb{C}_1\ \{o\}$ and ② $\vDash \{o\}\ \mathbb{C}_2\ \{q\}$. Consider $\mathbb{C}_1$. It is either `skip`, or another command.

   If it is `skip`, then by the single-step transition relation, the program $\mathbb{C}_1;\ \mathbb{C}_2$ takes an ID-labelled step to $\mathbb{C}_2$. By ① and the definition of the semantic triple judgement (definition 14), $\text{ID} \Vdash \{p\}\{o\}$. By ② and definition 14, the entire judgement $\vDash \{p\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \{q\}$ must hold.

   If $\mathbb{C}_1$ is not `skip`, then by the single-step labelled transition relation (definition 6), there is some $\mathbb{C}_1'$ such that $\mathbb{C}_1;\ \mathbb{C}_2$ transitions via some action labelled α to $\mathbb{C}_1';\ \mathbb{C}_2$. Moreover, by ① there is some $s \in \text{VIEWS}$ such that $\alpha \Vdash \{p\}\{s\}$ and $\vDash \{s\}\ \mathbb{C}_1'\ \{o\}$. The co-inductive hypothesis can now be applied to $\vDash \{s\}\ \mathbb{C}_1';\mathbb{C}_2\ \{q\}$, completing the result.

2. **If rule**: Assume ① $p \models \text{safe}(\text{B})$, ② $\vDash\ \{p \wedge \text{exprTrue}(\text{B})\}\ \mathbb{C}_1\ \{q\}$, and ③ $\vDash\ \{p \wedge \text{exprFalse}(\text{B})\}\ \mathbb{C}_2\ \{q\}$.

   By examination of the transitions of `if ( · ) · else ·`, three outcomes can occur. Expression B may evaluate to fault, and thus the program takes an $\mathcal{E}_{\lightning}(\text{B})$-labelled step to fault. It may evaluate to true, in which case an $\mathcal{E}_{\top}(\text{B})$-labelled step is taken to the program $\mathbb{C}_1$. It may evaluate to false, in which case an $\mathcal{E}_{\bot}(\text{B})$-labelled step is taken to $\mathbb{C}_2$. If the program does not reduce to fault, the state is not altered.

   Take the first outcome, $\mathcal{E}_{\lightning}(\text{B})$. By the fault-free property of reification (parameter 9), no reification can contain the fault state. Therefore, the expression evaluation on the pre-condition states must not fault (otherwise, the post-condition could not represent the result). That this does not occur is ensured by ① , and the boolean expression safety view (definition 11); by the definition of views as sets of instrumented heaps, and of entailment as set containment, any instrumented heap in p cannot reify to fault.

   Take the second outcome, $\mathcal{E}_{\top}(\text{B})$. For this to occur, on some heaps reified from p, B evaluates to true. The semantics state that $\texttt{if (B)}\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2, s \xrightarrow{\mathcal{E}_{\top}(\text{B})} \mathbb{C}_1, s$, where $s \in \lfloor p \rfloor$ and $(\![\text{B}]\!)(s) = \text{true}$ on those states in p. The semantic triple judgement requires us to show the existence of p_2 such that $\mathcal{E}_{\top}(\text{B}) \Vdash \{p\}\{p_2\}$ and $\vDash \{p_2\}\ \mathbb{C}_1\ \{q\}$. By the definition of conjunction and the action judgement, $\mathcal{E}_{\top}(\text{B}) \Vdash \{p\}\{p \wedge \text{exprTrue}(\text{B})\}$ holds, and $\vDash\ \{p \wedge \text{exprTrue}(\text{B})\}\ \mathbb{C}_1\ \{q\}$ holds by ② . Therefore, selecting $p_2 = p \wedge \text{exprTrue}(\text{B})$, both requirements of the triple judgement are fulfilled.

   The third outcome, $\mathcal{E}_{\bot}(\text{B})$, is similar, but using ③ .

3. **While rule**: Assume ① $p \models \text{safe}(\text{B})$ and ② $\vDash \{p \wedge \text{exprTrue}(\text{B})\}\ \mathbb{C}\ \{p\}$.

By examining the transitions for `while ( B )` $\mathbb{C}$, three outcomes can occur: B evaluates to true, B evaluates to false, or the evaluation of B faults. If it faults during evaluation, the program takes an $\mathcal{E}_{\lightning}(\text{B})$ step to the fault state. If it evaluates to true, then the program takes an $\mathcal{E}_{\top}(\text{B})$ step to $\mathbb{C}$; `while (B)` $\mathbb{C}$. If it evaluates to false, the program takes an $\mathcal{E}_{\bot}(\text{B})$ step to `skip`. If the evaluation does not fault, the state is not altered.

The first outcome, $\mathcal{E}_{\lightning}(\text{B})$, is discharged as in the `if` case.

For the second outcome, $\mathcal{E}_{\top}(\text{B})$, we must demonstrate some p_2 such that $\mathcal{E}_{\top}(\text{B}) \models \{p\}\{p_2\}$ and $\vDash \{p_2\}$ $\mathbb{C}$; `while (B)` $\mathbb{C}$ $\{q\}$ hold. Pick p_2 as $p \wedge \text{exprTrue}(\text{B})$. Then, $\mathcal{E}_{\top}() \models \{p\}\{p \wedge \text{exprTrue}(\text{B})\}$ holds by the action being $\mathcal{E}_{\top}(\text{B})$ and the exprTrue(B) conjunction. The semantic triple $\vDash \{p \wedge \text{exprTrue}(\text{B})\}$ $\mathbb{C}$; `while (B)` $\mathbb{C}$ $\{q\}$ holds by a combination of ② and the sequencing rule already proven.

For the third outcome, $\mathcal{E}_{\bot}(\text{B})$, the semantics takes an $\mathcal{E}_{\bot}(\text{B})$ step to `skip`, and we know the structured heap evaluates B to false. We must show that $\mathcal{E}_{\bot}(\text{B}) \models \{p\}\{p_2\}$ and $\vDash \{p_2\}$ `skip` $\{q\}$. Pick $p_2 = p \wedge \text{exprFalse}(\text{B})$. We must thus show $\mathcal{E}_{\bot}(\text{B}) \models \{p\}\{p \wedge \text{exprFalse}(\text{B})\}$, which is true by the definition of the action judgement in the $\mathcal{E}_{\bot}(\text{B})$ case. We must also show $\vDash \{p \wedge \text{exprFalse}(\text{B})\}$ `skip` $\{p \wedge \text{exprFalse}(\text{B})\}$, which is true by the skip rule (proposition 1).

4. **Parallel Composition rule**: We proceed by co-induction. Assume that ① $\vDash \{p_1\}$ $\mathbb{C}_1$ $\{q_1\}$ and ② $\vDash \{p_2\}$ $\mathbb{C}_2$ $\{q_2\}$.

   It must be the case that $\mathbb{C}_1$ || $\mathbb{C}_2$ takes an α-labelled transition for some α. By examination of the transition system, there are three cases: either $\mathbb{C}_1$ is skip, $\mathbb{C}_2$ is skip, or neither are skip. If either program is skip, the small-step semantics allow the program to take an ID. If either one is non-skip, the semantics allow that program to take a α step.

   If $\mathbb{C}_i =$ `skip`, $\mathbb{C}_j = \mathbb{C}'$ for $\{i, j\} \in \{1, 2\}$, and the semantics reduces the skip case, then ID $\models \{p_i\}\{q_i\}$ holds by the hypothesises, and thus ID $\models \{p_i * p_j\}\{q_i * p_j\}$ by the definition of the action judgement. $\vDash \{q_i * p_j\}$ $\mathbb{C}_j$ $\{q_i * q_j\}$ holds by the frame rule inference (which we prove sound shortly).

   If, say, $\mathbb{C}_1$ is not skip, it takes some α action to $\mathbb{C}_1'$ (the case for $\mathbb{C}_2$ is similar). By ① , $\alpha \models \{p_1\}\{p_1'\}$ for some p_1', and $\vDash \{p_1'\}$ $\mathbb{C}_1'$ $\{q_1\}$. The judgement $\vDash \{p_2\}$ $\mathbb{C}_2$ $\{q_2\}$ still holds, ergo $\vDash \{p_1' * p_2\}$ $\mathbb{C}_1'$ || $\mathbb{C}_2$ $\{q_1 * q_2\}$ holds by the co-inductive hypothesis.

To prove the abstraction rules:

1. **Frame rule**: We proceed by co-induction.

   Assume that ① $\vDash \{p\}\ \mathbb{C}\ \{q\}$. If $\mathbb{C} = \texttt{skip}$, then $\textsc{id} \Vdash \{p\}\{q\}$ by ① , and by the definition of the action judgement, $\textsc{id} \Vdash \{p * r\}\{q * r\}$ for all r. Ergo, $\vDash \{p * r\}\ \mathbb{C}\ \{q * r\}$, as required.

   If $\mathbb{C} \neq \texttt{skip}$, then by ① then for all action labels α and $s \in \lfloor p \rfloor$ such that $\mathbb{C}, s \xrightarrow{\alpha} \mathbb{C}', s'$ for some state s'. Moreover, there is some view p_2 such that $\alpha \Vdash \{p\}\{p_2\}$ and $s' \in \lfloor p_2 \rfloor$. By definition of the action judgement, $\alpha \Vdash \{p * r\}\{p_2 * r\}$ must hold, and $\vDash \{p_2\}\ \mathbb{C}'\ \{q\}$. Then, $\vDash \{p_2 * r\}\ \mathbb{C}'\ \{q * r\}$ holds by the co-inductive hypothesis.

2. **Disjunction rule**: To prove the rule of disjunction, we require two preliminary lemmas. First, that $*$ distributes over disjunction.

$$\begin{aligned} p * \bigvee\{q_i\}_{i\in I} &= p * \bigcup\{q_i\}_{i\in I} && \text{(Definition of disjunction)} \\ &= \bigcup\{p * q_i\}_{i\in I} && \text{(Pointwise definition of } *) \\ &= \bigvee\{p * q_i\}_{i\in I} && \text{(Definition of disjunction)} \end{aligned}$$

   Second, that disjunction is a morphism with respect to reification.

$$\begin{aligned} \lfloor \bigvee\{p_i\}_{i\in I} \rfloor &= \lfloor \bigcup\{p_i\}_{i\in I} \rfloor && \text{(Definition of disjunction)} \\ &= \bigcup \lfloor\{p_i\}\rfloor_{i\in I} && \text{(Pointwise definition of reification)} \end{aligned}$$

   Now assume the hypothesis for the rule: that ① for all $i \in I$, $\vDash \{p_i\}\ \mathbb{C}\ \{q_i\}$. Proceed by co-induction. If $\mathbb{C} = \texttt{skip}$ then by ① , for each i, $\textsc{id} \Vdash \{p_i\}\{q_i\}$. Unpacking this judgement, $\forall r \in \textsc{Views}. \lfloor p_i * r \rfloor \subseteq \lfloor q_i * r \rfloor$ holds. Call this result ② .

   We must show $\textsc{id} \Vdash \{\bigvee_{i\in I} p_i\}\{\bigvee_{i\in I} q_i\}$. This is equivalent to $\forall r \in \textsc{Views}.\ \lfloor r * \bigvee_{i\in I} p_i \rfloor \subseteq \lfloor r * \bigvee_{i\in I} q_i \rfloor$. By disjunction distribution, this is equal to $\forall r \in \textsc{Views}.\ \lfloor \bigvee_{i\in I} r * p_i \rfloor \subseteq \lfloor \bigvee_{i\in I} r * q_i \rfloor$. By disjunction morphism, this is equal to $\forall r \in \textsc{Views}.\ \bigcup_{i\in I} \lfloor r * p_i \rfloor \subseteq \bigcup_{i\in I} \lfloor r * q_i \rfloor$. For each element of the left hand side, its presence in the right hand side is ensured by ② and the commutativity of $*$.

   If $\mathbb{C} \neq \texttt{skip}$, then the program takes some α-labelled step to $\mathbb{C}'$ and for each p_i there is some r_i such that $\alpha \Vdash \{p_i\}\{r_i\}$ and $\vDash \{r_i\}\ \mathbb{C}'\ \{q_i\}$. By similar reasoning to the skip case, we have $\alpha \Vdash \{\bigvee_{i\in I} p_i\}\{\bigvee_{i\in I} r_i\}$, and the co-inductive hypothesis provides $\vDash \{\bigvee_{i\in I} r_i\}\ \mathbb{C}'\ \{\bigvee_{i\in I} q_i\}$.

3. **Consequence rule**: To prove the rule of consequence, we require the **action preservation** lemma: that the entailment $p \models q$ is no more than is allowed by the identity action, $\textsc{id} \Vdash \{p\}\{q\}$. For all $r \in \textsc{Views}$:

$$
\begin{array}{lrll}
 & p \models q & \triangleq p \subseteq q & \text{by definition} \\
\text{therefore } p \models q \implies & p * r & \subseteq q * r & * \text{ pointwise} \\
\text{therefore } p \models q \implies & \lfloor p * r \rfloor & \subseteq \lfloor q * r \rfloor & \lfloor \cdot \rfloor \text{ pointwise} \\
\text{therefore } p \models q \implies & & \textsc{id} \models \{p\}\{q\} & \text{definition of action judgement}
\end{array}
$$

Proceed by co-induction. Assume that, for all $p', q' \in \textsc{Views}$, ① $p \models p'$, ② $q' \models q$ and ③ $\vDash \{p'\}\ \mathbb{C}\ \{q'\}$.

Consider $\mathbb{C}$. If $\mathbb{C} = \texttt{skip}$, then by ③ $\textsc{id} \models \{p'\}\{q'\}$. We must show $\textsc{id} \models \{p\}\{q\}$. To see this, consider ① and the action preservation lemma, which implies that $\textsc{id} \models \{p\}\{p'\}$ holds. Note that $\textsc{id} \models \{p\}\{p'\}$ together with $\textsc{id} \models \{p'\}\{q'\}$ implies $\textsc{id} \models \{p\}\{q'\}$ by the definition of the action judgement. Using this, with ② and action preservation, produces $\textsc{id} \models \{p\}\{q\}$, as required.

If $\mathbb{C} \neq \texttt{skip}$ then by ③ , there is some action α, view p_2 and program $\mathbb{C}'$ such that $\alpha \models \{p'\}\{p'_2\}$ and $\vDash \{p'_2\}\ \mathbb{C}'\ \{q'\}$. We must show $\alpha \models \{p\}\{p'_2\}$, and $\vDash \{p'_2\}\ \mathbb{C}'\ \{q\}$. The first holds by ① , $\textsc{id} \models \{p\}\{p'\}$ together with $\textsc{id} \models \{p'\}\{q'\}$, and action preservation. The second holds by the co-inductive hypothesis.

□

In addition to these rules, the *rule of conjunction* is often included in Hoare reasoning.

Definition 15 (Semantic rule of conjunction)**.** Let I be some index set. If, for all $i \in I$, $\vDash \{p_i\}\ \mathbb{C}\ \{q_i\}$, then $\vDash \{\bigwedge_{i \in I} p_i\}\ \mathbb{C}\ \{\bigwedge_{i \in I} q_i\}$.

We do not include this rule amongst those of theorem 1 as it is not always sound. Our views represent sets of instrumented machine states, where the choice of instrumentation is not restricted. It is therefore possible that two valid proofs for the same program use *different choices* of instrumentation. In this case, there would be two valid triples $\vDash \{p_1\}\ \mathbb{C}\ \{q_1\}$ and $\vDash \{p_2\}\ \mathbb{C}\ \{q_2\}$. For the rule of conjunction to be sound, the triple $\vDash \{p_1 \wedge p_2\}\ \mathbb{C}\ \{q_1 \wedge q_2\}$ must be correct. However, recall that conjunction is defined as intersection of views (which are sets of instrumented heaps). It may be the case that $p_1 \wedge p_2$ describes a non-empty view, yet $q_1 \wedge q_2$ describes an empty view. The only programs for which such a post-condition is sound are divergent, so the rule can be used to generate unsound results. In chapter 3 we will see an example of instrumentation that is natural and useful, but for which the rule of conjunction does not hold.

We can ensure the soundness for the rule of conjunction with a condition on choices of actions and reification (and hence the action judgement). We call the property *primitive conjunctivity*.

Definition 16 (Primitive conjunctivity)**.** A set of actions labels ActionLbls (definition 5) have the **primitive conjunctivity property** if: for all $\alpha \in$ ActionLbls and sets of views p and q (definition 8) indexed by I, $\forall i \in I.\ \alpha \models \{p_i\}\{q_i\} \implies \alpha \models \{\bigwedge_{i\in I} p_i\}\{\bigwedge_{i\in I} q_i\}$.

This property ensures that, if an action was permitted in each of the potential conjuncts, the same action must be permitted in the resultant conjunction. If this property holds, then the semantic rule of conjunction is sound.

Lemma 1 (Soundness of the semantic rule of conjunction)**.** *Assume that primitive conjunctivity holds. Then, the rule of conjunction is sound.*

Proof. We proceed by co-induction. Assume that ① for all $i \in I$, $\vDash \{p_i\}\ \mathbb{C}\ \{q_i\}$. Furthermore, assume ② primitive conjunctivity (definition 16). If $\mathbb{C} = \texttt{skip}$ then by ① , for each p_i, the program takes an ID step, $\text{ID} \models \{p_i\}\{q_i\}$. By ② , $\text{ID} \models \{\bigwedge_{i\in I} p_i\}\{\bigwedge_{i\in I} q_i\}$ holds.

If $\mathbb{C} \neq \texttt{skip}$ then by ① , for each p_i there exists some α, r_i and $\mathbb{C}'$ such that both $\alpha \models \{p_i\}\{r_i\}$ and $\vDash \{r_i\}\ \mathbb{C}'\ \{q_i\}$. By ② , $\alpha \models \{\bigwedge_{i\in I} p_i\}\{\bigwedge_{i\in I} r_i'\}$ holds. By the co-inductive hypothesis, $\vDash \{\bigwedge_{i\in I} r_i'\}\ \mathbb{C}'\ \{\bigwedge_{i\in I} q_i\}$. □

2.3.3. Semantic consequence

The consequence rule in theorem 1 uses the entailment relation of definition 9. Informally, entailment $p_1 \models p_2$ is justified by the instrumented states of p_1 being a subset of those in p_2. We can show that everything that can be accomplished with p_1 can still be accomplished with p_2. We can define a similar relationship using the underlying machine states. The views framework calls this the *semantic consequence* relation. Semantic consequence is similar to entailment, but where entailment $p \models q$ works with relationships between the instrumented states in p and q, semantic consequence works with relationships between the uninstrumented machine state sets described by the views, $\lfloor p \rfloor$ and $\lfloor q \rfloor$.

Definition 17 (Semantic Consequence)**.** The **semantic consequence relation** $\preccurlyeq \subset$ Views × Views is defined such that:

$$(p, q) \in \preccurlyeq \iff \text{ID} \Vdash \{p\}\{q\}$$

An element $(p, q) \in \preccurlyeq$ is written write $p \preccurlyeq q$.

By unpacking the use of the action judgment in this definition, we see $p \preccurlyeq q \iff \forall r \in \text{VIEWS}.\ \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$. The relation allows an *view shift* from p to q, as long as the update does not change the machine level states under any frame. The embedding of r into the check is required to ensure that any change to the instrumentation does not invalidate instrumentation elsewhere. For example, consider fractional permissions. Without the "baked in" frame, the semantic consequence $\mathtt{x} \overset{0.5}{\to} v \preccurlyeq \mathtt{x} \overset{1}{\to} v$ would be allowed, as both views have the same reification. However, that update has invalidated many previously allowable frames, such as $\mathtt{x} \overset{0.25}{\to} v$.

We can replace the entailment relation of definition 9 with semantic consequence. Moreover, any relationship allowed by entailment is equally allowed by semantic consequence.

Lemma 2 (Semantic consequence is an entailment relation)**.** *The semantic consequence relation $\preccurlyeq$ is a sound replacement for the entailment relation $\models$ of definition 9. Moreover, it subsumes $\models$, $\models \subseteq \preccurlyeq$.*

Proof. Recall the proof of soundness of the consequence rule in theorem 1. The proof obligation required for soundness was that all actions were preserved by the entailment, $p \models q \implies \text{ID} \Vdash \{p\}\{q\}$. This is evidently true of the semantic consequence relation, which is defined as $\text{ID} \Vdash \{p\}\{q\}$. □

Comment 4 (Intuitionistic separation logic)**.** Recall the two reification operations for primitive heaps in examples 10 and 11. Under the first definition of reification, semantic consequence coincides with the entailment relation.

With the second definition, however, semantic consequence can do *more* than entailment. As this reification is "completing", it endows semantic consequence with a weakening ability, so that $p_1 * p_2 \preccurlyeq p_1$. This works because this reification will *provide* p_2:

$$
\begin{array}{ll}
& p_1 * p_2 \preccurlyeq p_1 \\
\iff & \forall r \in \textsc{Views}.\ \lfloor p_1 * p_2 * r \rfloor \subseteq \lfloor p_1 * r \rfloor \\
\iff & \quad \cup_{\mathbf{s}_1 \in p_1 * p_2 * r} \{s_1 \in \textsc{StructHeaps} \mid \forall a \in \mathrm{dom}(\mathbf{s}_1).\ s_1(a) = \mathbf{s}_1(a)\} \\
& \subseteq \cup_{\mathbf{s}_2 \in p_1 * r} \{s_2 \in \textsc{StructHeaps} \mid \forall a \in \mathrm{dom}(\mathbf{s}_2).\ s_2(a) = \mathbf{s}_2(a)\}
\end{array}
$$

The resultant set inclusion states that all heaps which are *consistent* with $p_1 * p_2 * r$ are contained within those which are consistent with $p_1 * r$, where consistent means that, for any shared addresses, the values stored in those cells are the same. The set inclusion holds as the domain of $p_1 * p_2 * r$ is strictly larger than that of $p_1 * r$ by the definition of primitive composition for primitive heaps.

That resource can be "forgotten about", $p * q \preccurlyeq p$, is one of the defining characteristics of intuitionsitic separation logic. Therefore, under semantic consequence, program reasoning with this choice of reification behaves intuitionistically. A small change to the reification function has large a effect on what can be proven.

2.3.4. The magic wand and weakest pre-conditions

The "magic wand" connective in separation logic forms a left adjoint of the separating conjunction with respect to entailment. That is, if $A * B \models C$, then $A \models B \mathrel{-\!\!*} C$. Informally, if $B \mathrel{-\!\!*} C$ holds, then the data described by it could be extended by B to result in data described by C. One common use of the magic wand is to express *parametric weakest pre-conditions*. The parametric weakest pre-condition of a program is some pre-condition p parametric in an assertion q, such that when the program is run in states satisfying the pre-condition, the resulting states satisfy q. Moreover, this pre-condition must be the *weakest*. The existence of parametric weakest pre-conditions for the axioms imply that a Hoare logic is complete for straight line code.

In our framework[1], the magic wand does not always carry the intuition from Separation Logic. The standard definition would be:

Definition 18 (Magic wand)**.** The **magic wand** $\mathrel{-\!\!*} : \textsc{Views} \times \textsc{Views}$ connective is defined as follows:

[1] And indeed, in views in general, where magic wand is sometimes not definable.

$$p \mathrel{-\!\!*} q \triangleq \bigvee \{o \in \textsc{Views} \mid o * p \models q\}$$

However, this does not account for view update performed by semantic consequence. We can recover the desired property with a connective identical to magic wand, but using semantic consequence rather than entailment. Due to our notation, we call this the "magic flower":

Definition 19 (Magic Flower)**.**

$$p \mathrel{-\!\!\!\preccurlyeq\!*} q \triangleq \bigvee \{o \in \textsc{Views} \mid o * p \preccurlyeq q\}$$

Lemma 3 (Magic flower adjointness)**.** *The magic flower is a left adjoint to semantic consequence,* $(p \mathrel{-\!\!\!\preccurlyeq\!*} q) * p \preccurlyeq q$.

Proof. By calculation.

$$\begin{array}{lll}
(p \mathrel{-\!\!\!\preccurlyeq\!*} q) * p & = \bigvee \{o \in \textsc{Views} \mid o * p \preccurlyeq q\} * p & \text{Expand flower definition} \\
& = \bigvee \{o \in \textsc{Views} * p \mid o * p \preccurlyeq q\} & * \text{ distributes over disjunction} \\
& = \bigcup \{o \in \textsc{Views} * p \mid o * p \preccurlyeq q\} & \text{Definition of disjunction} \\
& \preccurlyeq q & \text{Definition of set comprehension}
\end{array}$$

□

Aside from a short demonstration of weakest pre-condition construction, we will not use the magic wand nor flower in this thesis. Partially, this is because we have not found it useful outside of weakest pre-condition construction. Also, our long term research aim is *automation*, where proofs that use adjoints have been problematic.

2.4. Syntactic proof theory

The reasoning system presented in the last section is *semantic*, in the sense that inferences are made by analysing the underlying meaning of views. To aid reasoning, we now provide a *syntactic* proof theory. As far as possible, we use standard approaches to write *assertions* which act as the pre- and post-conditions of syntactic Hoare triples. These are manipulated

using a local Hoare logic with the standard syntactic deduction rules. We prove this system sound by interpreting it into the views of the previous section.

2.4.1. Assertion language

When constructing proofs, it is standard to link information between the pre- and post-conditions via *logical variables*. Logical variables are unrelated to program variables.

Parameter 13 (Logical variables and values)**.** Given a set of program variables PVars (definition 3), assume a non-empty set of **logical variables** LVars, ranged over by $\mathrm{x}, \mathrm{x}_1, \cdots, \mathrm{x}_n$, that is disjoint from program variables, $\mathrm{LVars} \cap \mathrm{PVars} = \emptyset$. Assume also a non-empty set of **logical values** LVals, ranged over by $V, V_1, \cdots, V_n$.

Logical variables are associated with logical values via *logical environments*.

Definition 20 (Logical environments)**.** Given a set of logical variables LVars and values LVals, the set of **logical environments** LEnvs, ranged over by $\Gamma, \Gamma_1, \cdots, \Gamma_n$, is the set of partial functions from logical variables to logical values:

$$\mathrm{LEnvs} = \{\Gamma \mid \Gamma : \mathrm{LVars} \rightharpoonup \mathrm{LVals}\}$$

To allow simple computation with logical variables and values, we include *logical expressions*. These expressions always include literals for logical values, logical variable lookup, and equality. They are extended with specific operations that aid reasoning in specific situations. Common extensions include set and string manipulation. The meaning of logical expressions is given by an *evaluation function*. This function can perform any interpretation, but must have standard behaviour with respect to literals and logical variables.

Parameter 14 (Logical expressions and interpretation)**.** Given a set of logical variables LVars and values LVals (parameter 13), and a set of logical environments LEnvs (definition 20), assume a set of **logical expressions** LExprs, ranged over by $\mathrm{E}, \mathrm{E}_1, \cdots, \mathrm{E}_n$, where $\mathrm{LVars} \cup \mathrm{LVals} \cup \{\mathrm{E}_1 = \mathrm{E}_2 \mid \mathrm{E}_1, \mathrm{E}_2 \in \mathrm{LExprs}\} \subseteq \mathrm{LExprs}$. Assume also a **logical expression evaluation** function $\langle\!|\cdot|\!\rangle(\cdot) : \mathrm{LExprs} \to \mathrm{LEnvs} \rightharpoonup \mathrm{LVals}$ that, for all $\Gamma \in \mathrm{LEnvs}, \mathrm{x} \in \mathrm{LVars}$, satisfies the equations:

$$
\begin{array}{ll}
(\!|V|\!)(\Gamma) & = V \\
(\!|\mathrm{x}|\!)(\Gamma) & = \Gamma(\mathrm{x}) \\
(\!|\mathrm{E}_1 = \mathrm{E}_2|\!)(\Gamma) & = (\!|\mathrm{E}_1|\!)(\Gamma) = (\!|\mathrm{E}_2|\!)(\Gamma)
\end{array}
$$

We now give a syntactic assertion language. By itself, an assertion in this language has no meaning, and is considered just a block of text with a certain format.

Definition 21 (Syntactic assertion language)**.** The **syntactic assertion language** Assts, ranged over by $P, Q, R, P_1, \cdots, P_n$, is defined by induction as follows:

$P ::=$	$P_1 * P_2$	Separating conjunction
	$\mid$ emp	Empty assertion
	$\mid P_1 \wedge P_2$	Conjunction
	$\mid P_1 \vee P_2$	Disjunction
	$\mid \neg P$	Negation
	$\mid$ **false**	Falsity
	$\mid P_1 \Rightarrow P_2$	Implication
	$\mid \exists \mathrm{x}.\ P$	Existential logical variable
	$\mid$ btrue(B)	Boolean truth filter
	$\mid$ bfalse(B)	Boolean false filter
	$\mid$ bsafe(B)	Boolean expression safety
	$\mid$ E	Logical expression
	$\mid -$	Domain specific assertions

A syntactic assertion is given meaning via an *interpretation* into a view. The interpretation function is a parameter of our framework, allowing for the user-defined behaviour of domain specific assertions. However, any choice for it must provide a standard set of behaviours for the assertions of definition 21. The interpretation also takes a logical environment, allowing the evaluation of logical expressions. Interpretation of logical expressions will result in the empty view (that is, false) if the logical expression itself cannot be evaluated.

Parameter 15 (Assertion interpretation)**.** Assume a **assertion interpretation** function $(\!|\cdot|\!)^{\cdot} : \textsc{LEnvs} \to \textsc{Assts} \to \textsc{Views}$ that satisfies the following equations

$$
\begin{array}{rl}
(\!|P * Q|\!)^{\Gamma} & = (\!|P|\!)^{\Gamma} * (\!|Q|\!)^{\Gamma} \\
(\!|\mathrm{emp}|\!)^{\Gamma} & = 0 \\
(\!|P \wedge Q|\!)^{\Gamma} & = (\!|P|\!)^{\Gamma} \wedge (\!|Q|\!)^{\Gamma} \\
(\!|P \vee Q|\!)^{\Gamma} & = (\!|P|\!)^{\Gamma} \vee (\!|Q|\!)^{\Gamma} \\
(\!|\neg P|\!)^{\Gamma} & = \neg (\!|P|\!)^{\Gamma} \\
(\!|\mathbf{false}|\!)^{\Gamma} & = \emptyset
\end{array}
\qquad
\begin{array}{rl}
(\!|P \Rightarrow Q|\!)^{\Gamma} & = (\textsc{InstHeaps} \setminus (\!|P|\!)^{\Gamma}) \cup (\!|Q|\!)^{\Gamma} \\
(\!|\exists \mathrm{x}.\ P|\!)^{\Gamma} & = \bigvee_{v \in \textsc{LVals}} (\!|P|\!)^{\Gamma[\mathrm{x} \mapsto v]} \\
(\!|\mathrm{btrue(B)}|\!)^{\Gamma} & = \mathrm{exprTrue(B)} \\
(\!|\mathrm{bfalse(B)}|\!)^{\Gamma} & = \mathrm{exprFalse(B)} \\
(\!|\mathrm{bsafe(B)}|\!)^{\Gamma} & = \mathrm{safe(B)} \\
(\!|\mathrm{E}|\!)^{\Gamma} & = \begin{cases} \textsc{InstHeaps} & \text{if } (\!|\mathrm{E}|\!)(\Gamma) = \mathbf{true} \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

The majority of definition 15 is standard. Of note is the existential logical variable quantification, $\exists \mathrm{x}.\ P$. This interprets to the infinite disjunction of the interpretations of P under *all possible bindings to* x. This is the semantic interpretation of existential quantification used in [22]: if there is *some* value one can assign to the logical variable such that P is satisfiable, it will be contained within this disjunct. The logical expression assertion evaluates logical expressions, and treats them as boolean outcomes. It is useful for the analysis of logical variables, so that (for example) $P \wedge \mathrm{v} = 5$ is a valid assertion, equivalent to P if v is 5, and equivalent to $false$ otherwise.

It is useful to determine if one syntactic assertion entails another. We thus lift the entailment relation of definition 9 to syntactic assertions.

Definition 22 (Entailment assertion)**.** Given the entailment relation $\models$ (definition 9), assertions Assts (definition 21) and logical environments LEnvs (definition 20), the **entailment assertion** $\models_{\Gamma} \subseteq \textsc{Assts} \times \textsc{Assts}$ is defined as:

$$(P, Q) \in \models_{\Gamma} \iff \forall \Gamma \in \textsc{LEnvs}.\ (\!|P|\!)^{\Gamma} \models (\!|Q|\!)^{\Gamma}$$

An element $(P, Q) \in \models_{\Gamma}$ is written $P \models_{\Gamma} Q$.

Similarly, we will want to determine when two assertions semantically entail each other.

Definition 23 (Semantic consequence assertion)**.** Given the semantic consequence relation $\preccurlyeq$ (definition 17), assertions Assts (definition 21) and logical environments LEnvs (definition 20), the **semantic consequence assertion** $\preccurlyeq_{\Gamma} \subset \textsc{Assts} \times \textsc{Assts}$ is defined as:

$$(P, Q) \in \preccurlyeq_\Gamma \iff \forall \Gamma \in \textsc{LEnvs}.\ (\!|P|\!)^\Gamma \preccurlyeq (\!|Q|\!)^\Gamma$$

An element $(P, Q) \in \preccurlyeq_\Gamma$ is written $P \preccurlyeq_\Gamma Q$.

2.4.2. Local Hoare reasoning

Just as definition 12 built semantic Hoare triples with views, we build *syntactic Hoare triples* with assertions.

Definition 24 (Syntactic Hoare triples)**.** Given a set of assertions Assts (definition 21) and programs Programs (definition 4), the set of **syntactic Hoare triples** SynTrip is defined as:

$$\textsc{SynTrip} \triangleq \textsc{Assts} \times \textsc{Programs} \times \textsc{Assts}$$

We can then give syntactic axioms, mirroring the semantic axioms of definition 11.

Parameter 16 (Syntactic axioms)**.** Given a set of atomic commands AtomicCmds (parameter 5), assume a set of **syntactic axioms** SynAxioms $\subseteq$ Assts $\times$ AtomicCmds $\times$ Assts, such that, for each $\mathrm{c} \in$ AtomicCmds, there exists some $P, Q \in$ Assts with $(P, \mathrm{c}, Q) \in$ SynAxioms.

Example 17 (Primitive heap logic)**.** To give a syntactic presentation of our primitive heap example (example 16), we must first provide logical variables, values and expressions (parameters 13 and 14 respectively). For primitive heaps, we require the ability to link pre- and post-conditions via logical variables. For this, we need only use set of program values for logical values, LVals = PVals. The set of logical expressions is the minimal set given by the definition. The interpretation is the smallest function satisfying the definitional equations.

We can then define the domain specific syntactic assertions for primitive heaps with variables as resource. There are two standard cells: a variable-as-resource cell, and the flat heap cell. There is also an *expression evaluation assertion*. This links the result of evaluating an expression to the result of evaluating a logical expression. It

will be used to determine the outcome of evaluating expressions.

The assertions are defined as: for all $\mathtt{x} \in \text{PVars}, \mathrm{E}, \mathrm{E}_1, \mathrm{E}_2 \in \text{LExprs}, e \in \text{Exprs}$

$\mathtt{x} \to \mathrm{E}$	Variable cell
$\mathrm{E}_1 \mapsto \mathrm{E}_2$	Flat heap cell
$e \Rightarrow \mathrm{E}$	Expression evaluation cell

Recalling the view syntax of example 16, their interpretations are:

$$\begin{aligned}
(\!|\mathtt{x} \to \mathrm{E}|\!)^{\Gamma} &= \mathtt{x} \mapsto (\!|\mathrm{E}|\!)(\Gamma) \\
(\!|\mathrm{E}_1 \mapsto \mathrm{E}_2|\!)^{\Gamma} &= (\!|\mathrm{E}_1|\!)(\Gamma) \mapsto (\!|\mathrm{E}_2|\!)(\Gamma) \\
(\!|e \Rightarrow \mathrm{E}|\!)^{\Gamma} &= e \Rightarrow (\!|\mathrm{E}|\!)(\Gamma)
\end{aligned}$$

It will often be useful to describe a variable where we do not know the value. We can derive such an assertion as:

$$\mathtt{x} \to - \triangleq \exists \mathrm{X}.\mathtt{x} \to \mathrm{X}$$

The syntactic axioms are: for all $\mathtt{x} \in \text{PVars}$, $e \in \text{Exprs}$, $\mathrm{X}, \mathrm{Y}, \mathrm{Z} \in \text{LVars}$, $P \in \text{Assts}$

$\{(\mathtt{x} \to \mathrm{X} * P) \wedge e \Rightarrow \mathrm{Y}\}$	`x := e`	$\{\mathtt{x} \to \mathrm{Y} * P\}$
$\{\mathtt{x} \to \mathrm{X}\}$	`x := alloc()`	$\{\exists \mathrm{X}, \mathrm{Y}.(\mathtt{x} \to \mathrm{X} * \mathrm{X} \mapsto \mathrm{Y})\}$
$\{\mathrm{X} \mapsto \mathrm{Y} * (P \wedge e \Rightarrow \mathrm{X})\}$	`free(e)`	$\{P\}$
$\{\mathrm{X} \mapsto \mathrm{Y} * (P \wedge e_1 \Rightarrow \mathrm{X} \wedge e_2 \Rightarrow \mathrm{Z})\}$	`[`e_1`] := `e_2	$\{\mathrm{X} \mapsto \mathrm{Z} * P\}$
$\{\mathrm{Y} \mapsto \mathrm{Z} * ((\mathtt{x} \to \mathrm{X} * P \wedge e \Rightarrow \mathrm{Y}))\}$	`x := [e]`	$\{\mathrm{Y} \mapsto \mathrm{Z} * \mathtt{x} \to \mathrm{Z} * P\}$

Every syntactic axiom must correspond to some semantic axiom (definition 11) in that for *every* interpretation of the syntactic axiom, we can find a semantic axiom that justifies it.

Parameter 17 (Syntactic small axiom soundness)**.** Assume that for each syntactic small axiom $(P, \mathrm{C}, Q) \in \text{SynAxioms}$ and for any logical environment $\Gamma \in \text{LEnvs}$, there exists some semantic axiom $((\!|P|\!)^{\Gamma}, \mathrm{C}, (\!|Q|\!)^{\Gamma}) \in \text{Axioms}$.

The axioms for primitive heaps 17 can be seen as sound simply by interpreting them into their underlying views.

Program proofs are constructed using *syntactic proof rules*. These rules are standard from local Hoare logics, but extend the consequence rule into a *semantic consequence rule* by using the semantic consequence relation of definition 17. This presentation is used by [22], and allows for proofs that change the instrumentation associated with assertions.

Definition 25 (Syntactic proof rules)**.** Given assertions Assts (definition 21) and the syntactic entailment and semantic consequence relations $\models_\Gamma$ and $\preccurlyeq_\Gamma$ (definition 22 and definition 23 respectively), there are two classes of proof rules: those driven by analysis of program syntax, and those driven by analysis of assertion syntax. They are: for all assertions $P, Q, R, O \in \text{Assts}$

Program syntax rules

Skip rule:

$$\frac{}{\vdash \{P\}\ \texttt{skip}\ \{P\}}$$

Axiom rule:

$$\frac{(P, \text{c}, Q) \in \text{SynAxioms}}{\vdash \{P\}\ \text{c}\ \{Q\}}$$

Sequencing rule:

$$\frac{\vdash \{P\}\ \mathbb{C}_1\ \{O\} \quad \vdash \{O\}\ \mathbb{C}_2\ \{Q\}}{\vdash \{P\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \{Q\}}$$

If rule:

$$\frac{P \models_\Gamma \text{bsafe}(\text{B}) \quad \vdash \{P \wedge \text{btrue}(\text{B})\}\ \mathbb{C}_1\ \{Q\} \quad \vdash \{P \wedge \text{bfalse}(\text{B})\}\ \mathbb{C}_2\ \{Q\}}{\vdash \{P\}\ \texttt{if (B)}\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2\ \{Q\}}$$

While rule:

$$\frac{P \models_\Gamma \text{bsafe}(\text{B}) \quad \vdash \{P \wedge \text{btrue}(\text{B})\}\ \mathbb{C}\ \{P\}}{\vdash \{P\}\ \texttt{while (B)}\ \mathbb{C}\ \{P \wedge \text{bfalse}(\text{B})\}}$$

Parallel rule:

$$\frac{\vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\vdash \{P_1 * P_2\}\ \mathbb{C}_1 \mid\mid \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

Assertion syntax rules

Frame rule:

$$\frac{\vdash \{P\}\ \mathbb{C}\ \{Q\}}{\vdash \{P * R\}\ \mathbb{C}\ \{Q * R\}}$$

Disjunction rule:

$$\frac{\vdash \{P_1\}\ \mathbb{C}\ \{Q_1\} \quad \vdash \{P_2\}\ \mathbb{C}\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ \mathbb{C}\ \{Q_1 \vee Q_2\}}$$

Consequence rule:

$$\frac{P \preccurlyeq_\Gamma P' \quad \vdash \{P'\}\ \mathbb{C}\ \{Q'\} \quad Q' \preccurlyeq_\Gamma Q}{\vdash \{P\}\ \mathbb{C}\ \{Q\}}$$

Conjunction rule*:

$$\frac{\vdash \{P_1\}\ \mathbb{C}\ \{Q_1\} \quad \vdash \{P_2\}\ \mathbb{C}\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ \mathbb{C}\ \{Q_1 \wedge Q_2\}}$$

Existential elimination rule:

$$\frac{\vdash \{P\}\ \mathbb{C}\ \{Q\}}{\vdash \{\exists \mathrm{x}.\ P\}\ \mathbb{C}\ \{\exists \mathrm{x}.\ Q\}}$$

* The conjunction rule is only valid if the underlying view system has the primitive conjuctivity property (definition 16).

We now show that any syntactic triple derived with our proof system is a valid triple.

Theorem 2 (Soundness of the syntactic rules)**.** *Given logical environments* LEnvs *(definition 20), assertions* Assts *(definition 21) and programs* Programs *(definition 4), for all logical environments* $\Gamma \in$ LEnvs*, assertions* $P, Q \in$ Assts *and programs* $\mathbb{C} \in$ Programs*,* $\vdash \{P\}\ \mathbb{C}\ \{Q\}$ *implies* $\vDash \{(\!|P|\!)^\Gamma\}\ \mathbb{C}\ \{(\!|Q|\!)^\Gamma\}$.

Proof. The proof is by induction on the structure of the derivation. It is straightforward, as most of the syntactic rules have an equivalent semantic rule in theorem 1. We demonstrate only one example of this case. We also give the existential case, which has no associated semantic rule.

Sequential case: For any logical environment $\Gamma \in$ LEnvs, the assertions P, Q and O interpret to some views p, q and o respectively. By the premises, $\vdash \{P\}\ \mathbb{C}_1\ \{O\}$ and $\vdash \{O\}\ \mathbb{C}_2\ \{Q\}$ hold. By the inductive hypothesis $\vDash \{p\}\ \mathbb{C}_1\ \{o\}$ and $\vDash \{o\}\ \mathbb{C}_2\ \{q\}$ hold. By the semantic inference rules, $\vDash \{p\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \{q\}$ holds. This is equal to $\vDash \{(\!|P|\!)^\Gamma\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \{(\!|Q|\!)^\Gamma\}$, hence $\vdash \{P\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \{Q\}$ is valid.

Existential case: Assume that, for all $\Gamma \in$ LEnvs, $\vdash \{P\}\ \mathbb{C}\ \{Q\}$. By the inductive hypothesis, $\vDash \{(\!|P|\!)^\Gamma\}\ \mathbb{C}\ \{(\!|Q|\!)^\Gamma\}$ holds. By the assumption and inductive hypothesis, it must be that, for all Γ and V, both $(\!|P|\!)^{\Gamma[\mathrm{x}\mapsto V]}$ and $(\!|Q|\!)^{\Gamma[\mathrm{x}\mapsto V]}$ hold. Therefore, by the semantic conjunction rule, $\vDash \{\bigvee_{V \in \mathrm{LVals}} (\!|P|\!)^{\Gamma[\mathrm{x}\mapsto V]}\}\ \mathbb{C}\ \{\bigvee_{V \in \mathrm{LVals}} (\!|Q|\!)^{\Gamma[\mathrm{x}\mapsto V]}\}$ holds. The

conclusion pre-condition $\exists \mathrm{x}.\ P$ inteprets as $\bigvee_{v \in \mathrm{LVAL_S}} (\!|P|\!)^{\Gamma[\mathrm{x} \mapsto V]}$, and similarly for Q. The result follows. □

2.4.3. Additional language rules

The programming language we have defined is simple, supporting only a minimal set of standard commands. This is sufficient for demonstrating our reasoning and for giving examples, as the techniques of this thesis are focused on axiomatic specifications for libraries. The specific choices of language features (such as local variables and functions) are orthogonal to our contributions. To ease the presentation of examples, we will occasionally assume some of these richer features, but will never require them for our library specifications.

We expect our techniques to apply to richer languages without difficulty. Partially, this is due to our use of views [22]. The views system provides a general pattern for developing reasoning systems. For example, results from the views paper show how any separation algebra can be directly lifted into a view system. Existing separation logic work, such as that on Java [56] or JavaScript [34] should be easily recast into a view. Our library reasoning can then be directly applied. In the long term, we will use this approach to provide richer automated reasoning, by linking with separation logic reasoning tools such as Verifast [45].

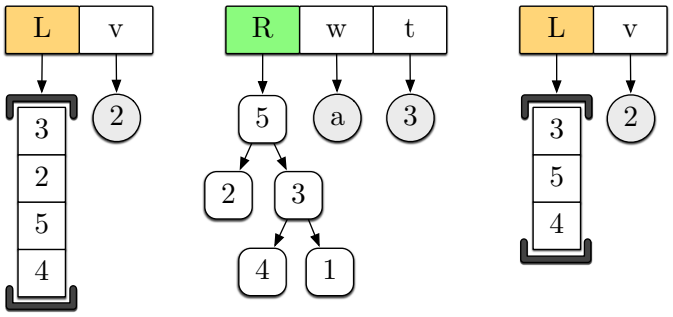
2.5. Summary

This chapter has introduced *addressed value views*, a specialisation of the views framework [22]. We will build on this foundation in the rest of our thesis, using it to provide the core program logic for structural separation logic. There are few novel contributions in this chapter, as it merely specialises existing work to our specific needs. However, two minor contributions are included:

1. **Views for a WHILE language**: Section 2.3 gives reasoning rules for the standard WHILE language features, `if` and `while`. As opposed to [22], which was given in terms of more abstract language with non-deterministic choice and iteration, this allows views-style reasoning about standard example programs.
2. **Syntactic proof theory**: Section 2.4 offers a syntactic proof theory for views, adding a standard syntactic assertion language and associated syntactic Hoare rules. This enables standard rules such as existential elimination, which are not given in the views publications.

3. Structural separation logic

We now turn to programming with and reasoning about libraries that manipulate *abstract* data. Rather than creating data structures such as lists and trees, by imposing shapes on heap cells via inductive or abstract predicates, we define structured heaps that store rich data in single cells. For example, by picking structured heap values that are "entire lists", heaps can store a list or tree at a single address. The first and last diagrams below store a whole list at address L, whilst the centre stores a whole tree at address R (the other cells are simple variables):



Along with these rich values, we define collections of atomic commands that manipulate these values directly. For example, the command `remove(2)`, which removes a value from a list, transforms the above left-hand list heap into the right-hand list. Our choices of structured heaps and commands are designed to give imperative machines that match the mental model programmers have of libraries. We will give two library examples in section 3.1, working with *lists* and *unranked trees* of integers.

Reasoning about these libraries requires more than standard separation. The interesting structure is not in the *domain* of the heap, but rather in the *values*. Standard techniques from separation logic (section 1.2.1) are not sufficient, as we want to split the structured values into smaller sub-data. The data splitting approaches of context or segment logic (section 1.2.3) are also insufficient, as we want to retain the simple shape of, and reasoning associated with, heaps. This chapter introduces *structural separation logic* to enable fine-grained reasoning about these structures whilst retaining the natural heap reasoning style

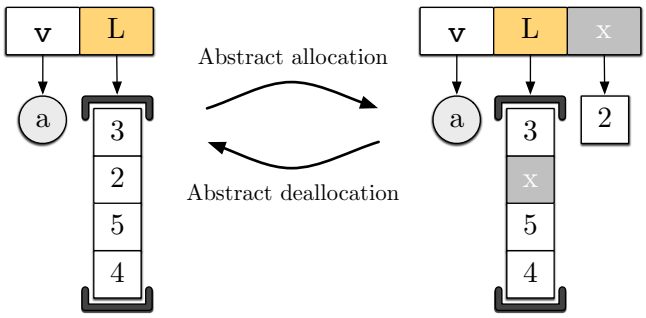


Figure 3.1.: Abstract allocation and deallocation for the sub-data 2 of the list stored at L.

of separation logic.

Structural separation logic consists of two key contributions. The first is *abstract heaps*. Abstract heaps are an instrumentation for structured heaps that enables fine-grained reasoning about structured heaps. They are similar in shape to structured heaps, but allow the reasoning to create *abstract heap cells* at *structural addresses*. When fine grained access to *sub-data* of some rich value is desired, that sub-data can be "cut out" from its current location, and promoted to a fresh abstract heap cell. This process, which exists only to facilitate program reasoning, *abstract allocation and deallocation*, is our second contribution.

Abstract allocation and deallocation is illustrated in figure 3.1. On the left-hand side is a structured heap without any abstract heap cells. If we wanted to isolate the list element 2 for analysis, we can apply abstract allocation. This process finds a new *abstract address* $\mathbf{x}$, cuts the element 2 from the list, and places it a new abstract heap cell at address $\mathbf{x}$. The cutting is achieved using an approach similar to multi-holed contexts (section 1.2.3), and so leaves behind a *body address* in the place of the element 2. Once the analysis is complete, this body address enables *abstract deallocation*, in which the abstract heap cell is destroyed, and its contents replace the body address $\mathbf{x}$, a process we call *compression*. Abstract allocation is like machine allocation, in that we do not know exactly *which* structural address will be picked, but we do know there will be one available to choose regardless of the addresses used in any frame. Like machine allocation, this ensures that no possible frames can be invalidated by the creation of abstract heap cells.

Abstract allocation enables the isolation of fine grained data in abstract heap cells. These abstract cells can then be used to give small footprints to commands that operate on rich data. Consider a command `remove(x)` that removes `x` from a list. Via abstract heap cells we can give an axiom to the command that describes the precise effect of

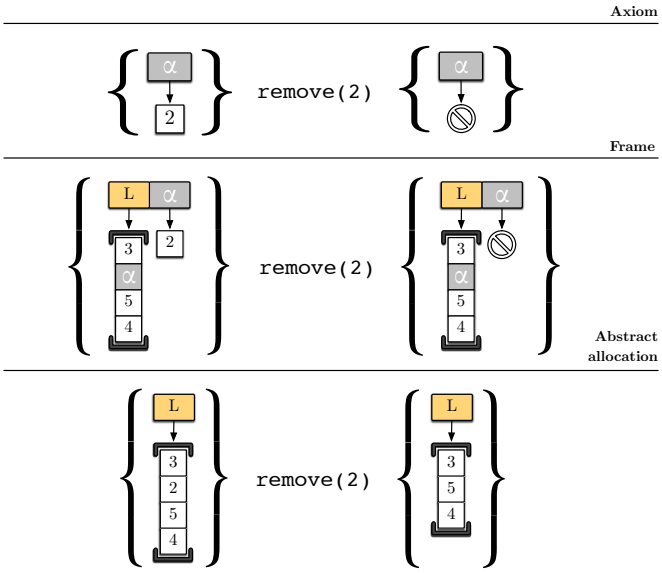


Figure 3.2.: Derivation using abstract allocation.

removing a single list element. Proofs using abstract heaps can describe the effect of commands only on the data used, such as in figure 3.2, in which the logical variable α is used to contain the address used for abstract allocation. The proof first uses abstract allocation to isolate the list element 2 in an abstract heap cell. The resultant abstract heap behaves exactly like a normal heap, so we can use the standard frame rule to set aside the rest of the list at L. The axiom for `remove` can then be applied, the frame restored, and abstract deallocation used to merge the (now empty) data at α back into the list.

The abstract heap cells and body addresses, despite being only instrumentation (and hence invisible to program commands), do create different instrumented structured heaps. The various abstract heaps in figure 3.3 are all distinct, as they have differing uses of structural addresses to cut up the data. However, once all the abstract heap cells are compressed, they do represent the same structured heap. As our framework uses the view system, each of these heaps are related by *semantic consequence* (section 2.3.3). The structural addressing, being instrumentation, can be updated by semantic consequence. This allows abstract allocation and deallocation to be achieved in the proof theory by the semantic consequence rule of the program logic.

Structural separation logic can be created for many types of structured data, and can be used to give small axioms for many types of libraries. The data and libraries we

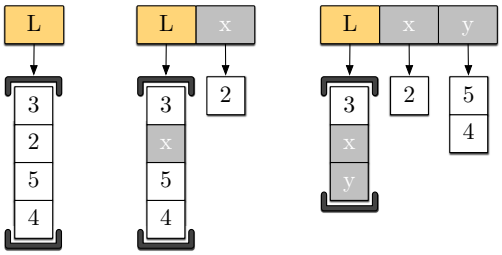


Figure 3.3.: Different abstract heaps for the same list. Each list heap (and tree heap) is identical in terms of content, but has different *structural addressing*. The heaps are all *semantic consequences* of each other.

demonstrate in this chapter are simple, allowing us to focus on the technical details of structural separation logic. Here, we introduce two example libraries, those of *lists* and *trees*. The tree case is similar to the list case in terms of constructing heaps, but has a quite different set of commands. Later, we use the techniques from this chapter to reason about larger libraries such as DOM (chapter 4) and the POSIX file system (chapter 6).

3.1. Structured data libraries

We begin by constructing imperative machines for *lists* and *trees*. We give an abstract representation of the data, and library commands that work directly on this abstract representation.

3.1.1. List library

We consider finite lists of unique positive integers, defined in two parts: *partial lists* are sequences of numbers; and *lists* are partial lists wrapped in delimiters that indicate they are complete. We split the definition this way to emphasise the *structured* nature of lists as being composed of many partial lists. It will also prove convenient for defining the atomic commands. We use unique elements to ensure each command can operate on a guaranteed distinct element, which reduces the verbosity of our library commands[1].

Definition 26 (Lists)**.** The set of lists Lists, ranged over by $l, l_1, \cdots, l_n$ is defined by induction as: for all $i \in \mathbb{N}^+$

[1]This choice is purely to simplify the presentation, and is not a requirement of structural separation logic

$$\begin{array}{rll} pl ::= & i \mid pl_1 \otimes pl_2 \mid \varnothing & \text{Partial lists} \\ l ::= & [pl] & \text{Lists} \end{array}$$

where $\otimes$ is associative with left and right identity $\varnothing$, and each list contains only distinct numbers. Lists are equal up to the properties of $\otimes$.

Example lists are $[1 \otimes 5 \otimes \varnothing]$, $[6 \otimes 2]$ and $[\varnothing]$. The $\otimes$ operator being associative but not commutative ensures the lists are ordered. The identity property means that $[1 \otimes 5 \otimes \varnothing]$ is equal to $[1 \otimes 5]$. Note that $[5 \otimes 5]$ is *not* one of these lists, as the elements are not unique.

List modules typically allow programmers to work with many lists, creating, updating and destroying them as necessary. For simplicity, we initially consider a module that works on just a single list and will consider the multiple list case in section 3.5.2. We therefore build *list heaps* by extending the variables as resource heaps (introduced in example 2) with a single *list address* mapping to a complete list structure.

Definition 27 (List heaps)**.** Given sets of program variables PVars and program values PVals (definition 3), and lists Lists, and assuming a single heap list address L, where $\mathrm{L} \notin \mathrm{PVars}$, the set of **list machine heaps** ListHeaps, ranged over by $lh, lh_1, \cdots, lh_n$, is defined as:

$$\mathrm{ListHeaps} \triangleq \{lh \mid lh : (\mathrm{PVars} \xrightarrow{\mathrm{fin}} \mathrm{PVals}) \sqcup (\{\mathrm{L}\} \to \mathrm{Lists})\}$$

Notice that the list portion of the heap is a total function from a singleton domain; there is always exactly one list in the heap. An example list heap instance is given in figure 3.4.

Consider the commands that might manipulate a list. Common operations are appending an element to the back of a list, removing an element, searching the list for an element, and finding elements adjacent to other elements. Rather than aim for the most realistic collection of commands here, we pick a small set designed to demonstrate a range of structural manipulations. We choose the following list commands, and in chapters 4 and 6 we will examine more realistic libraries.

1. `append(i)`: Appends the value of variable `i` to the list, making it the last element. Faults if the value of `i` is already in the list, or if it is not a positive integer.
2. `remove(i)`: Removes the value of variable `i` from the list. Faults if the value of `i` is not in the list.

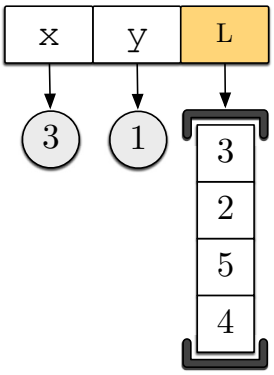


Figure 3.4.: An instance of a list heap (definition 27), storing the variables `x` and `y` and the list L with value $[3 \otimes 2 \otimes 5 \otimes 4]$.

3. `i := getFirst()`: Assigns to `i` the first element of the list, or 0 if the list has no elements.
4. `j := getRight(i)`: Assigns to `j` the first element in the list that is directly to the right of the value contained in `i`. If the value of `i` is the last element of the list, assigns 0. Faults if the value of `i` is not in the list.

Recall that in primitive heaps (example 4), faults occur only when the heap contained insufficient addresses (that is, a missing variable or unallocated heap cell). These list commands can also fault based upon the *values* contained within heap addresses.

Definition 28 (List update commands)**.** Given a set of program variable names PVars (definition 3) and program expressions Exprs (example 5), the **atomic commands of the list update language** are those of the variable system (example 2), and the following additional commands: for all $\mathtt{i}, \mathtt{j} \in$ PVars, $e \in$ Exprs

$$
\begin{aligned}
\text{C} := \;& \texttt{append}(e) \\
\mid \;& \texttt{remove}(e) \\
\mid \;& \texttt{i := getFirst()} \\
\mid \;& \texttt{j := getRight}(e)
\end{aligned}
$$

Given partial lists PartialLists (definition 26), the actions of the list commands are: for all $i, j \in \mathbb{N}^+, pl, pl_1, pl_2 \in$ PartialLists

$$[\!(\texttt{append}(e))\!](s) \triangleq \begin{cases} \{s[\mathrm{L} \mapsto [pl \otimes (\!|e|\!)(s)]]\} & \text{if } s(\mathrm{L}) = [pl], (\!|e|\!)(s) \in \mathbb{N} \\ & \quad \nexists pl_1, pl_2.\ pl = pl_1 \otimes (\!|e|\!)(s) \otimes pl_2 \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\!(\texttt{remove}(e))\!](s) \triangleq \begin{cases} \{s[\mathrm{L} \mapsto [pl_1 \otimes pl_2]]\} & \text{if } s(\mathrm{L}) = [pl_1 \otimes (\!|e|\!)(s) \otimes pl_2] \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\!(\texttt{i := getFirst()})\!](s) \triangleq \begin{cases} \{s[\mathtt{i} \mapsto i]\} & \text{if } s(\mathrm{L}) = [i \otimes pl] \\ \{s[\mathtt{i} \mapsto 0]\} & \text{if } s(\mathrm{L}) = [\varnothing] \end{cases}$$

$$[\!(\texttt{j := getRight}(e))\!](s) \triangleq \begin{cases} \{s[\mathtt{j} \mapsto j]\} & \text{if } s(\mathrm{L}) = [pl_1 \otimes (\!|e|\!)(s) \otimes j \otimes pl_2] \\ \{s[\mathtt{j} \mapsto 0]\} & \text{if } s(\mathrm{L}) = [pl \otimes s(\mathtt{i})] \\ \{\lightning\} & \text{otherwise} \end{cases}$$

Notice how each command operates on the entire list heap cell, yet does not interact with the majority of the list. For example, the successful `remove` case analyses the list as $[pl_1 \otimes (\!|e|\!)(s) \otimes pl_2]$, and results in the list $[pl_1 \otimes pl_2]$. The partial lists pl_1 and pl_2 are unchanged, as elements are unique so there an be no further instances of e, thus their structure is unanalysed. A normal programmer's intuition for these commands is that they act *only* on the element(s) being operated on; the result of evaluating e in the remove case. This is not obviously reflected in the operational actions, but will be clearly visible in the axiomatic semantics.

This small set of commands can be used to write implementations for many library commands we do not axiomatise. The specifications we will derive for these implementations could equally well have been included as axioms for the commands. These implementations are both a sanity check and shortcut. Given a command implementation, if our axioms allow us to derive the specification we would expect, then we have more confidence that our subset is useful (as it proves useful programs) and complete (in that we have sufficient primtive commands to achieve all needed manipulations of the data structure). Moreover, if our reasoning is sound, there is no need to prove the soundness of additional axioms for these commands. We take this approach in both our DOM and POSIX file system examples (chapters 4 and 6 respectively).

Command implementations are also a convenient source of demonstrations for our reasoning. A command `i := getLast()` can be implemented by using `getFirst`, and re-

```
i := getLast() ≜ local j {
  i := 0;
  j := getFirst();
  while (j ≠ 0)
    i := j;
    j := getRight(i)
}
```

```
b := contains(k) ≜ local i {
  i := getFirst();
  b := false;
  while (i ≠ 0)
    if (i = k)
      b := true;
    else
      skip;
    i := getRight(i);
}
```

Figure 3.5.: Example programs for list commands.

peatedly applying `getRight` until 0 is returned. Similarly, a command `contains(k)` that determines if an element stored in variable `i` exists in the list can be given. Both of these examples are presented in figure 3.5.

In these examples, we assume a simple local variable construction, `local p1, ..., pn { C }`, which restricts the scope of variables `p1` through `pn` to the program $\mathbb{C}$. We justify using such a language extension in section 2.4.3, and it is used only to aid readability of the examples. It does not change the expressivity of the system in a meaningful fashion.

3.1.2. Tree library

The techniques we used to build the imperative machine for lists can also be used to build a similar machine for trees. We consider trees as finite, unranked trees consisting of *nodes* uniquely identified by positive integers.

Definition 29 (Trees)**.** The set of **trees** TREES, ranged over by $t, t_1, \cdots, t_n$, is defined inductively as follows: for all $i \in \mathbb{N}^+$

$$t ::= i[t] \mid t_1 \otimes t_2 \mid \varnothing$$

where $\otimes$ is associative with left and right identity $\varnothing$, and each tree contains unique node identifiers. Trees are equal up to the properties of $\otimes$.

Some example trees are given in figure 3.6. Notice that a tree may have many root nodes, and that their children are ordered. For convenience, we often write $i[\varnothing]$ as just i.

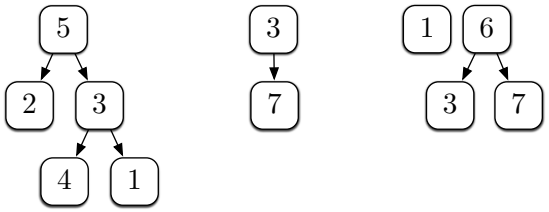


Figure 3.6.: The example trees $5[2[\varnothing] \otimes 3[4[\varnothing] \otimes 1[\varnothing]]]$, $3[7[\varnothing]]$ and $1[\varnothing] \otimes 6[3[\varnothing] \otimes 7[\varnothing]]$.

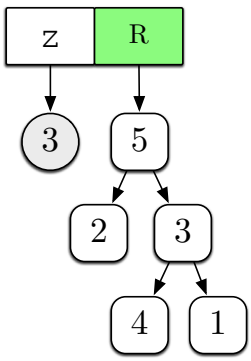


Figure 3.7.: An instance of a tree heap (definition 30), storing the variable z, and the tree $5[2 \otimes 3[4 \otimes 1]]$.

As in lists, we work with a tree module that manipulates just a single tree. This is a surprisingly realistic scenario. For example, web browsers typically work with a single DOM tree representing the web-page. There is also only a single file system on many operating systems.

Definition 30 (Tree heaps)**.** Given program variables PVars (definition 3) and trees Trees (definition 29), and assuming a single heap tree address $\mathrm{R} \notin \mathrm{PVars}$, the set of **tree heaps** TreeHeaps, ranged over by $th, th_1, \cdots, th_n$, is defined as:

$$\mathrm{TreeHeaps} \triangleq \{th \mid th : (\mathrm{PVars} \xrightarrow{\text{fin}} \mathrm{PVals}) \sqcup (\{\mathrm{R}\} \to \mathrm{Trees})\}$$

Notice that the tree at address R may contain many root nodes; there is a distinction between roots of the tree, and addresses in the heap. One can think of this as being two different namespaces: entire trees are stored in the heap at an address, and, once in the

tree values, there may be many roots. A tree heap instance is given in figure 3.7.

As in lists, we pick commands designed to demonstrate different types of structural manipulations, rather than the most realistic library. The command `createNode` demonstrates how the tree can grow in size by the creation of nodes. The commands `appendNode` and `removeSubtree` demonstrate structural manipulations that can *fault* based on specific properties of the tree being analysed. The `getChild` command demonstrates analysis of the tree children. Their behaviors are:

1. `nID := createNode()`: Creates a node using a fresh identifier, without a parent, and assigns the new node identifier to the variable `nID`.
2. `appendNode(pID, cID)`: Moves the subtree identified by the value of `cID` to be the last child of the node identified by `pID`. Faults if either `pID` or `cID` do not identify nodes within the tree, or if the node identified by `cID` is an ancestor of the node identified by `pID`.
3. `nID := getChild(pID, i)`: Assigns the identifier of the `i` $++\,1^{\text{th}}$ child of the node identified by `pID` to `nID`. If `i` is negative, or the node has fewer than `i` + 1 children, assigns 0 to `nID`. Faults if `nID` does not identify a node.
4. `removeSubtree(nID)`: Deletes the subtree identified by `nID` from the tree. Faults if `nID` does not identity a node within the tree.

The notion of partial lists was useful in giving actions to the list commands in definition 28. For example, the `remove(i)` command analysed the list as $[pl_1 \otimes s(\mathtt{i}) \otimes pl_2]$, and gave the result $[pl_1 \otimes pl_2]$. Notice that pl and pl_2 were just "carried" between the input and result. Their specific properties are ignored, so the command will work with any choice for them. It will be similarly convenient to give actions to tree commands using *single holed tree contexts*. As trees are structurally more complex, contexts will allow us to extract sub-trees from a tree whilst ignoring the surrounding data. Tree contexts are intuitively "trees with a hole", and have an associated composition function that fills the hole to form a complete tree. Note that, by construction, there will be exactly one context hole in a single-holed tree context.

Definition 31 (Single-holed tree contexts)**.** Given a set of trees Trees (definition 29), the set of **single-holed tree contexts** TreeContexts, ranged over by $\mathbf{ct}, \mathbf{ct}_1, \cdots, \mathbf{ct}_n$, is defined inductively as follows: for all $i \in \mathbb{N}^+, t \in$ Trees

$$\mathbf{ct} ::= i[\mathbf{ct}] \mid \mathbf{ct} \otimes t \mid t \otimes \mathbf{ct} \mid -$$

where $-$ is the context hole, $\otimes$ is associative with left and right identity $\varnothing$, and each tree context contains unique node identifiers. Tree contexts are equal up to the properties of $\otimes$.

Definition 32 (Tree context composition)**.** Given sets of single-holed tree contexts TreeContexts (definition 31) and trees Trees (definition 29), the **tree context composition** function $comp$: TreeContexts $\to$ Trees $\rightharpoonup$ Trees is defined by induction on the structure of tree contexts as:

$$\begin{aligned} comp(i[\mathbf{ct}], t) &\triangleq i[comp(\mathbf{ct}, t)] \\ comp(\mathbf{ct} \otimes t_1, t_2) &\triangleq comp(\mathbf{ct}, t_1) \otimes t_2 \\ comp(t_1 \otimes \mathbf{ct}, t_2) &\triangleq t_1 \otimes comp(\mathbf{ct}, t_2) \\ comp(-, t) &\triangleq t \end{aligned}$$

If the result would have duplicate node identifiers, $comp$ is undefined. The instance $comp(\mathbf{ct}, t)$ is written $\mathbf{ct} \circ t$.

The action for `getChild` in definition 34 will require the ability to calculate the length of a list of siblings.

Definition 33 (Tree length)**.** Given a set of trees Trees (definition 29), the **tree length** function len : Trees $\to \mathbb{N}$, returning the number of nodes in a tree, is defined by induction on trees in the standard way.

The tree module consists of the following atomic commands and associated actions, defined in terms of the above functions.

Definition 34 (Tree update commands)**.** Given a set of program variable names PVars (definition 3) and program expressions Exprs (example 5), the atomic commands of the tree update library are those of the variable system (example 2), the following additional commands: for all $\mathtt{nID}, \mathtt{cID}, \mathtt{pID}, \mathtt{t}, \mathtt{i} \in$ PVars, $e \in$ Exprs

$$
\begin{aligned}
\text{C} ::= \quad & \texttt{nID := createNode()} \\
\mid\ & \texttt{appendNode(pID, cID)} \\
\mid\ & \texttt{nID := getChild(pID, } e\texttt{)} \\
\mid\ & \texttt{removeSubtree(nID)}
\end{aligned}
$$

Given sets of trees Trees (definition 29) and single-holed tree contexts TreeContexts (definition 31), the actions of the tree commands are as follows: for all $c, i, p, \in \mathbb{N}^+, t, t_1, t_2, t_3 \in \textsc{Trees}, \mathbf{ct}, \mathbf{ct}_1, \mathbf{ct}_2 \in \textsc{TreeContexts}$

$$
[\![\texttt{nID := createNode()}]\!](s) \triangleq \left\{ s[\mathrm{R} \mapsto t_1, \texttt{nID} \mapsto i] \;\middle|\; \begin{array}{l} i \in \mathbb{N}^+, t_1 = s(\mathrm{R}) \otimes i[\varnothing], \\ \nexists \mathbf{ct}, t_2.\ s(\mathrm{R}) = \mathbf{ct} \circ i[t_2] \end{array} \right\}
$$

$$
[\![\texttt{appendNode(pID, cID)}]\!](s) \triangleq \begin{cases} \{s[\mathrm{R} \mapsto \mathbf{ct}_2 \circ p[t_2 \otimes c[t_1]]]\} & \text{if } \begin{array}{l} s(\texttt{pID}) = p, \\ s(\texttt{cID}) = c, \\ s(\mathrm{R}) = \mathbf{ct}_1 \circ c[t_2] \\ \mathbf{ct}_1 \circ \varnothing = \mathbf{ct}_2 \circ p[t_2] \end{array} \\ \{\lightning\} & \text{otherwise} \end{cases}
$$

$$
[\![\texttt{nID := getChild(pID, } e\texttt{)}]\!](s) \triangleq \begin{cases} \{s[\texttt{nID} \mapsto c]\} & \text{if } \begin{array}{l} s(\texttt{pID}) = p, (\![e]\!)(s) = i \\ s(\mathrm{R}) = \mathbf{ct} \circ p[t_1 \otimes c[t_2] \otimes t_3]], \\ i = len(t_1) \end{array} \\ \{s[\texttt{nID} \mapsto 0]\} & \text{if } \begin{array}{l} s(\texttt{pID}) = p, (\![e]\!)(s) = i, \\ s(\mathrm{R}) = \mathbf{ct} \circ p[t], \\ i < 0 \vee i \geq len(t) \end{array} \\ \{\lightning\} & \text{otherwise} \end{cases}
$$

$$
[\![\texttt{removeSubtree(nID)}]\!](s) \triangleq \begin{cases} \{s[\mathrm{R} \mapsto \mathbf{ct} \circ \varnothing\} & \text{if } s(\texttt{nID}) = i, s(\mathrm{R}) = \mathbf{ct} \circ i[t] \\ \{\lightning\} & \text{otherwise} \end{cases}
$$

The actions in definition 34 are largely straightforward. The only complexity is within `appendNode`, which must check that the node `cID` is not an ancestor of the node `pID`. If it were, performing the append would create a cyclic "tree". This property is ensured by

```
c := count(n) ≜ local d {
  c := 0;
  d := getChild(n, c);

  while (d ≠ 0)
    c := c + 1;
    d := getChild(n, c)
}
```

Figure 3.8.: Example tree program.

first splitting into the new child, and a surrounding context via $\mathbf{ct}_1 \circ c[t_2]$. The context $\mathbf{ct}_1$ is re-analysed by extracting the parent (having first filled the child's position with empty), $\mathbf{ct}_1 \circ \varnothing = \mathbf{ct}_2 \circ p[t_2]$. This ensures that we cannot find the parent as a descendant of the child, which is equivalent to the property we need.

As in the list module, we can define additional commands in terms of the given set. A command `count(n)` that returns the number of children of node `n` is given in figure 3.8.

3.2. Abstract heaps

The list and tree libraries (definition 28 and definition 34 respectively) allow programs that manipulate the abstract structures without concern for the implementation. Our use of partial lists and single-holed contexts show that whilst the command actions are on *entire* heap cells, they do not access most of the data in a cell. The majority of the commands can be tracked to only the *local* sub-data they use. We now define *abstract heaps*, which provide a principled approach to reasoning about updates on local sub-data. Abstract heaps extend the structured heaps of an imperative machine (parameter 2) with *structural addresses*. These addresses enable *addressable values* and *abstract heap cells*.

Addressable values are instrumentation for machine values that allow them to be cut-up and re-joined on structural addresses via *compression* and *decompression*. In figure 3.9, decompression uses address $\mathbf{x}$ cuts a list into *super-data*, containing the body address $\mathbf{x}$, and *sub-data* that was removed via the cut. With decompression, we can localise sub-data on which we want to act. The cut data can be re-joined by compression on $\mathbf{x}$.

We then use structural addresses to add *abstract heap cells* to structured heaps. This extends the heap domain beyond just MachineAddrs, allowing it to store sub-data in these "virtual" cells, using structural addresses as *abstract address*. Abstract heap cells are purely instrumentation, invisible to the program, and exist to allow fine-grained access

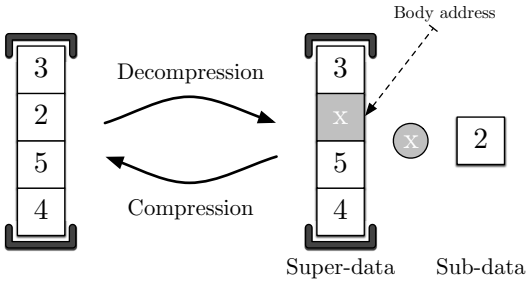


Figure 3.9.: Compression and decompression

to sub-data during program proofs. We call the process of creating and removing abstract heap cells *abstract allocation*, and it is illustrated in figure 3.1. Abstract allocation "picks" some fresh structural address not in use in the heap as either a body nor heap address, uses decompression to cut the data on the chosen address, and promotes the resulting sub-data to a heap cell. Abstract deallocation reverses this process.

Abstract heaps are built upon a choice of imperative machine. We first demonstrate their construction via our list and tree examples. We give body addresses and compression to the data structures, creating *abstract lists* and *abstract trees*. We then use this instrumentation to give an informal notion of how abstract heaps can be built in these cases. Having introduced the concepts with lists and trees, we will then generalise body addresses and compression to data in general. Using this general presentation, we will formalise abstract heaps.

We require a choice of structural addresses to act as both body and abstract heap addresses in our examples. We will use the same set of structural addresses for all data.

Definition 35 (Structural addresses)**.** Assume a countably infinite set of **structural addresses** StructAddrs, ranged over by $\mathbf{x}, \mathbf{y}, \mathbf{z}, \cdots$.

3.2.1. Abstract lists and trees

Abstract lists

To define abstract heaps for lists, we must give an approach for compression and decompression of list data using structural addresses. We call addressable list data *abstract lists*, and define them using multi-holed contexts. We use structural addresses to label the context holes, and the holes themselves act as body addresses.

Definition 36 (Abstract lists)**.** Given a set of structural addresses StructAddrs (definition 35), the set of **abstract partial lists** AbsPartialLists, ranged over by $\mathbf{pl}, \mathbf{pl}_1, \cdots, \mathbf{pl}_n$, and the set of **abstract lists** AbsLists, ranged over by $\mathbf{l}, \mathbf{l}_1, \cdots, \mathbf{l}_n$ is defined by induction as: for all $i \in \mathbb{N}, \mathbf{x} \in$ StructAddrs

$$\begin{array}{rll} \mathbf{pl} ::= & i \mid \mathbf{pl}_1 \otimes \mathbf{pl}_2 \mid \varnothing \mid \mathbf{x} & \text{Abstract partial lists} \\ \mathbf{l} ::= & [\mathbf{pl}] & \text{Abstract lists} \end{array}$$

where $\otimes$ is associative with left and right identity $\varnothing$, and each list contains only distinct numbers and body addresses. Abstract (partial) lists are equal up to the properties of $\otimes$.

Notice we define a set of AbsLists and AbsPartialLists. This is because, in abstract heaps, whilst the root address L will store abstract lists, the abstract heap cells will store only partial lists.

It will be useful to extract the set of body addresses present in an abstract list.

Definition 37 (Addresses of abstract lists)**.** Given a set of abstract lists AbsLists (definition 36) using structural addresses StructAddrs (definition 35), the **addresses function** addrs : (AbsLists $\cup$ AbsPartialLists) $\to \mathcal{P}$(StructAddrs) is defined by induction as:

$$\begin{array}{ll} \text{addrs}(i) & \triangleq \emptyset \\ \text{addrs}(\mathbf{pl}_1 \otimes \mathbf{pl}_2) & \triangleq \text{addrs}(\mathbf{pl}_1) \cup \text{addrs}(\mathbf{pl}_2) \\ \text{addrs}(\varnothing) & \triangleq \emptyset \\ \text{addrs}(\mathbf{x}) & \triangleq \{\mathbf{x}\} \\ \text{addrs}([\mathbf{pl}]) & \triangleq \text{addrs}(\mathbf{pl}) \end{array}$$

We define compression for abstract lists using standard context substitution on body addresses. As addresses are unique, at most one will be removed by compression.

Definition 38 (Compression for abstract lists)**.** Given a set of abstract lists AbsLists (definition 36) using structural addresses StructAddrs (definition 35), the **compression function** comp : StructAddrs $\to$ (AbsLists $\cup$ AbsPartialLists) $\to$ (AbsLists $\cup$ AbsPartialLists) $\rightharpoonup$ (AbsLists $\cup$ AbsPartialLists) is defined by induction on the structure of abstract lists as:

$$
\begin{array}{ll}
\mathrm{comp}(\mathbf{x}, i, \mathbf{pl}) & \triangleq i \\
\mathrm{comp}(\mathbf{x}, \mathbf{pl}_1 \otimes \mathbf{pl}_2, \mathbf{pl}_3) & \triangleq \mathrm{comp}(\mathbf{x}, \mathbf{pl}_1, \mathbf{pl}_3) \otimes \mathrm{comp}(\mathbf{x}, \mathbf{pl}_2, \mathbf{pl}_3) \\
\mathrm{comp}(\mathbf{x}, \varnothing, \mathbf{pl}) & \triangleq \varnothing \\
\mathrm{comp}(\mathbf{x}, \mathbf{y}, \mathbf{pl}) & \triangleq \begin{cases} \mathbf{pl} & \text{if } \mathbf{x} = \mathbf{y} \\ \mathbf{y} & \text{otherwise} \end{cases} \\
\mathrm{comp}(\mathbf{x}, [\mathbf{pl}_1], \mathbf{pl}_2) & \triangleq [\mathrm{comp}(\mathbf{x}, \mathbf{pl}_1, \mathbf{pl}_2)]
\end{array}
$$

If the result would not be a member of AbsLists, it is undefined. The application $\mathrm{comp}(\mathbf{x}, \mathbf{l}_1, \mathbf{l}_2)$ is written $\mathbf{l}_1 \circledast_{\mathbf{x}} \mathbf{l}_2$. We overload comp to also act on partial lists.

The compression operation implicitly defines decompression. Consider three abstract lists, $\mathbf{l}_1, \mathbf{l}_2$ and $\mathbf{l}_3$, where $\mathbf{l}_1 \circledast_{\mathbf{x}} \mathbf{l}_2 = \mathbf{l}_3$. We can read the equality in both directions, stating that $\mathbf{l}_1$ and $\mathbf{l}_2$ *compress* using $\mathbf{x}$ to form $\mathbf{l}_3$, and also that $\mathbf{l}_3$ *decompresses* via $\mathbf{x}$ into $\mathbf{l}_1$ and $\mathbf{l}_2$. This gives a notion of *super-data* and *sub-data*. As $\mathbf{l}_3$ decompresses to $\mathbf{l}_1$ and $\mathbf{l}_2$, we consider $\mathbf{l}_1$ to be *super-data* of $\mathbf{l}_2$ and, symmetrically, $\mathbf{l}_2$ to be *sub-data* of $\mathbf{l}_1$.

Abstract heaps for lists

Decompression allows us to cut-up our data and build *abstract heaps*. Abstract heaps are an instrumented version of structured heaps, and form the models of our local reasoning for rich structured data. We will formally define them in section 3.4. However, they are quite intuitive objects. For this list case, an abstract heap is an extension of the list heap (definition 27), adding *abstract heap cells*. We define abstract heaps in two parts. First, we give *pre-abstract heaps* for lists, which extend the structured heaps for lists by using abstract lists as values, and allowing *abstract heap cells* containing abstract partial lists.

$$
\mathbf{h}_l : (\mathrm{PVars} \xrightarrow{\mathrm{fin}} \mathrm{PVals}) \sqcup (\{\mathrm{L}\} \rightharpoonup \mathrm{AbsLists}) \sqcup (\mathrm{StructAddrs} \xrightarrow{\mathrm{fin}} \mathrm{AbsPartialLists})
$$

Notice that, whilst L maps to AbsLists, the structural addresses map to partial lists. Via abstract deallocation, we shall create abstract heap cells using abstract partial lists only. We can never remove the entire list from L, so it can store only abstract lists.

Pre-abstract lists are not yet usable for reasoning, as they may not represent *consistent* data. Figure 3.10 gives the three typical problems. Pre-abstract list heap (a) contains two $\mathbf{x}$ body addresses. This is *ambiguous*, as it connects the same sub-data to two different super-data. This is nonsense, as data cannot belong to two super-data simultaneously- this

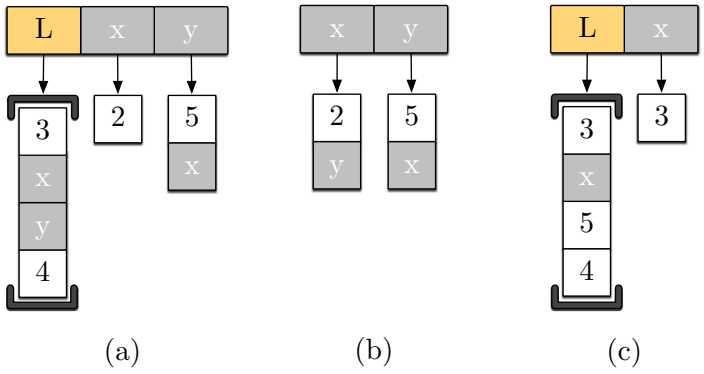


Figure 3.10.: Invalid abstract list heaps.

would create a cyclical data structure. Pre-abstract list heap (b) has structural address **x** present as a body address in the value at address **y**. Similarly, the address **y** is present as a body address in the value at address **x**. This is a *cycle*, where the addressing connects sub-data to itself. Data cannot be super-data of itself, so this is nonsense. Finally, pre-abstract list heap (c) correctly uses structurally addresses, but after abstract cell **x** is compressed, will represent an invalid list containing two 3 elements. Structural addresses must always compress to valid instances of the data structure.

The *abstract list heaps* are those pre-abstract list heaps without these problems. They represent structured list heaps that have used structural addresses to sensibly cut-up lists, so that by repeatedly apply abstract deallocation, they return to a normal structured list heap. However, these abstract heaps may be *partial*, where some of the sensibly cut-up data is absent. In these cases, an abstract heap is useful if we can find *some* additional resource that, when added, results in a abstract heap that can be deallocated to a normal heap.

Figure 3.11 represents three valid abstract lists heaps. Abstract list heap (a) is a normal structured list heap, using no structured addressing. Such heaps are always good abstract heaps. Abstract list heap (b) represents one abstract allocation, where address **x** has been used. By deallocating **x**, we return to a normal heap. Abstract list heap (c) represents a single abstract heap cell. Even though there is nothing to deallocate, it is still a valid abstract heap, as we can find resource (e.g. the cell L from (b)) that, when added, would allow it to collapse to a structured heap. These three heaps are, respectively, the models of the pre-conditions at each step of figure 3.2.

As abstract heaps are the models of our reasoning, we must provide an interpretation

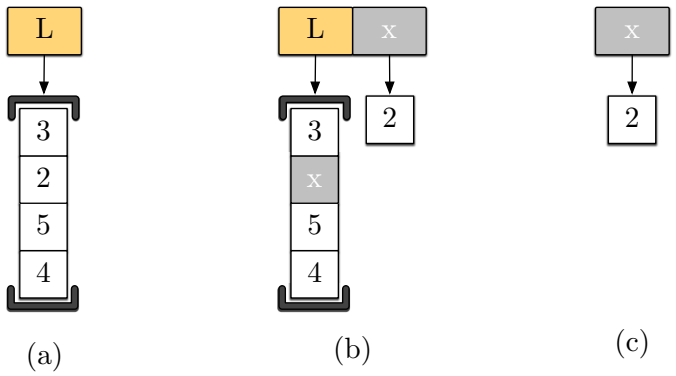


Figure 3.11.: Valid abstract list heaps.

of them as machine heaps via the *reification* process used in the framework (chapter 2). We use a reification of *completion*. The completion process takes a heap, augments it with additional data, them uses abstract deallocation repeatedly until no further addresses can be deallocated. If the result is a structured heap, it is kept as a result. The completion of pre-abstract heap $\mathbf{x} \mapsto 2$ is given in figure 3.12.

Abstract trees

Just as with lists, body addresses can be added to trees to enable the creation of abstract tree heaps. We will follow this pattern for all future examples: given a data structure, we add structural addresses to the structure, define an addrs function, and give a notion of compression comp.

Definition 39 (Abstract trees)**.** Given a set of structural addresses StructAddrs (definition 35), the set of **abstract trees** AbsTrees, ranged over by $\mathbf{t}, \mathbf{t}_1, \cdots, \mathbf{t}_n$, is defined inductively as follows: for all $i \in \mathbb{N}^+$, $\mathbf{x} \in$ StructAddrs

$$\mathbf{t} ::= i[\mathbf{t}] \mid \mathbf{t}_1 \otimes \mathbf{t}_2 \mid \varnothing \mid \mathbf{x}$$

where $\otimes$ is associative with left and right identity $\varnothing$, and each tree contains only distinct tree identifiers and body addresses. Abstract trees are equal up to the properties of $\otimes$.

The addresses and compression functions for abstract trees are defined similarly to the list case.

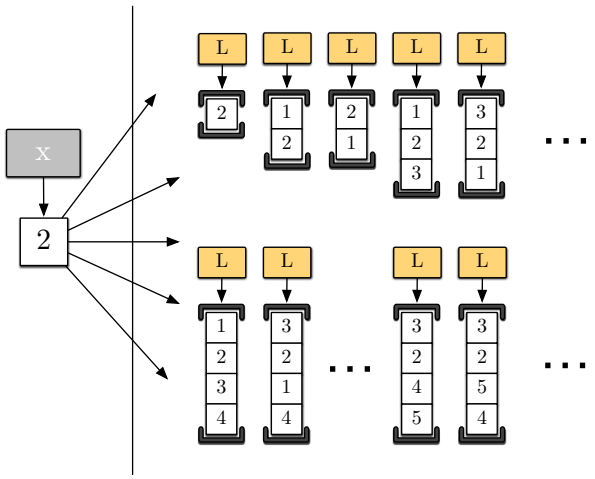


Figure 3.12.: Part of the completion for an abstract list heap $\mathbf{x} \mapsto 2$. The set is infinite, consisting of all possible lists containing element 2. We do not consider program variables.

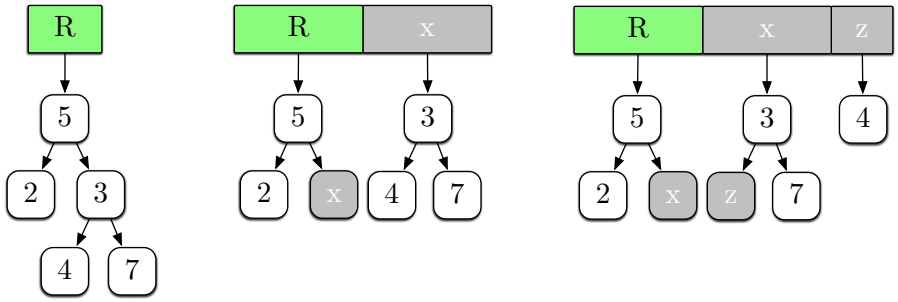
Definition 40 (Addresses of abstract trees)**.** Given a set of abstract trees AbsTrees (definition 39) on structural addresses StructAddrs (definition 35), the **addresses function for abstract trees** addrs : AbsTrees $\to \mathcal{P}$(StructAddrs) is defined similarly to definition 37.

Definition 41 (Compression for abstract trees)**.** Given a set of abstract trees AbsTrees (definition 39) on structural addresses StructAddrs (definition 35), the **compression function for abstract trees** comp : StructAddrs $\to$ AbsTrees $\to$ AbsTrees $\rightharpoonup$ AbsTrees is defined similarly to definition 38. As there, the instance $\text{comp}(\mathbf{x}, \mathbf{t}_1, \mathbf{t}_2)$ is written $\mathbf{t}_1 \circledast_{\mathbf{x}} \mathbf{t}_2$.

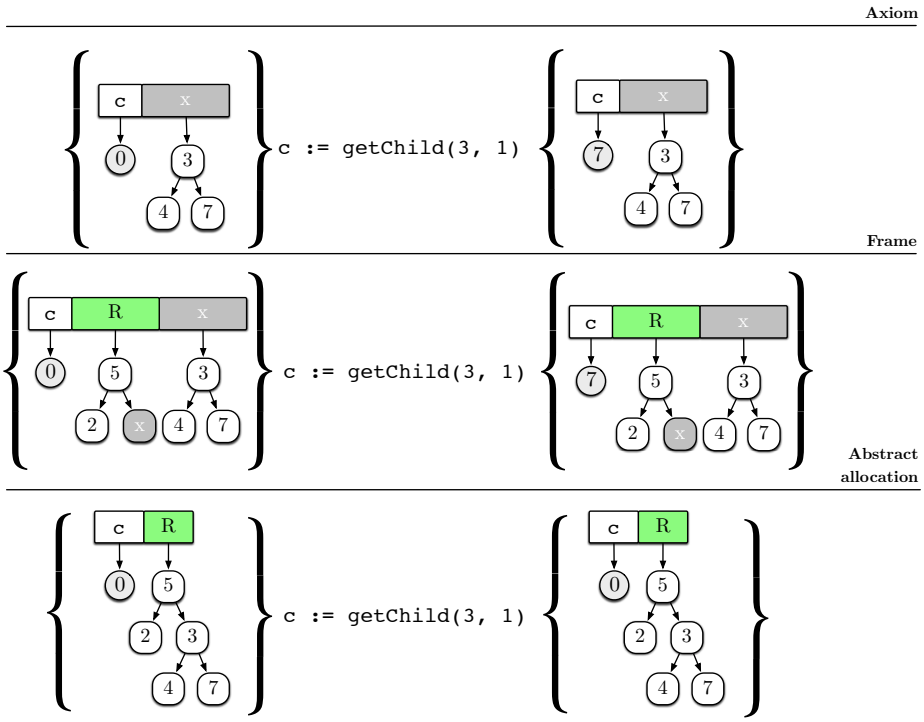
Abstract tree heaps are very similar to the abstract list heaps introduced earlier. The type of the pre-abstract tree heaps is:

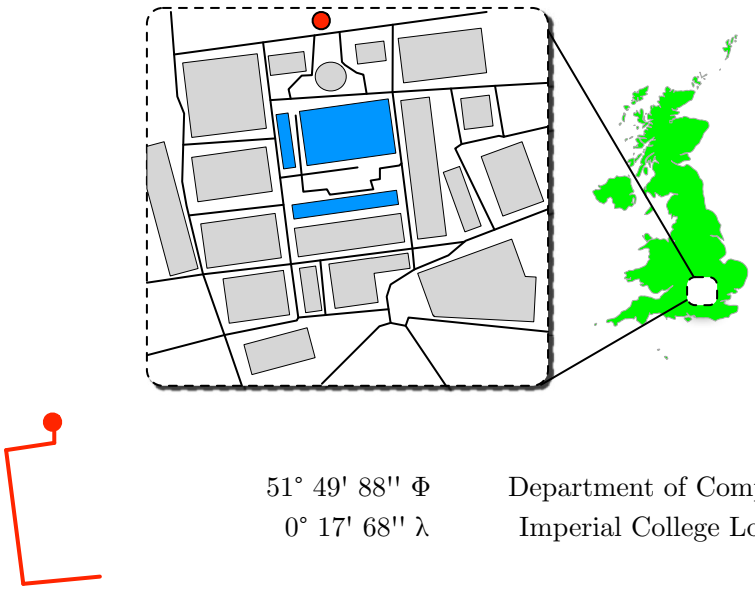
$$\mathbf{h}_t : (\text{PVars} \xrightarrow{\text{fin}} \text{PVals}) \sqcup (\{\text{R}\} \rightharpoonup \text{AbsTrees}) \sqcup (\text{StructAddrs} \xrightarrow{\text{fin}} \text{AbsTrees})$$

The abstract tree heaps are, as in the list case, the pre-abstract tree heaps with well-formedness conditions. They must not have ambiguity nor cycles in their structural addressing, no can the addressing hide invalid data. Three example abstract tree heaps, all representing the same structured heap, are:



With these heaps, we can use abstract allocation to create local-reasoning style proofs, such as the following for `c := getChild(3, 1)`.





51° 49' 88'' Φ
0° 17' 68'' λ

Department of Computing
Imperial College London

(a) (b) (c)

Figure 3.13.: A map of the Imperial College South Kensington campus area, and its position within the United Kingdom. Below the map are three methods of finding the department of Computing: (a) some street directions from a known location, (b) latitude and longitude, and (c) the department's unique name.

3.2.2. Formalising abstract heaps

We have introduced abstract lists and abstract trees, using *compression* on structural *body addresses* to enable the cutting up of rich data. This allowed us to also introduce abstract heaps, where the structural *abstract addresses* act as instrumentation atop structured heaps. We now generalise these concepts, giving a notion of compression for arbitary rich data, and using it to formalise the construction of abstract heaps.

To add compression to lists and heaps, we used multi-holed contexts. The context holes acted as body addresses within data, and context application was compression. We now show that, with minor changes, the *multi-holed context algebras* of [21] can be used to give a generally useful notion of addressing in data. We will not focus on the context-like nature of compression and holes, instead treating them as structural addresses in data. We explain why via an analogy of *maps*.

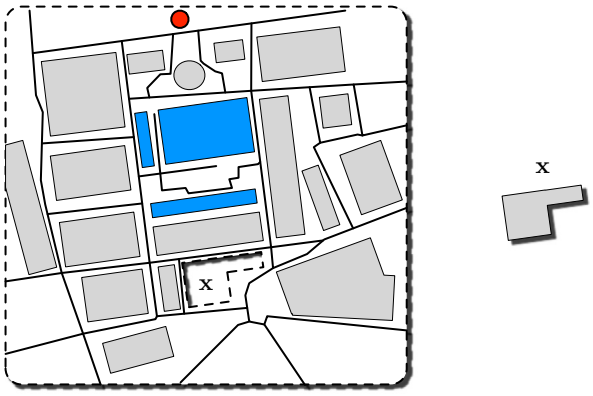
Figure 3.13 contains a map of the area around the Department of Computer Science

at Imperial College London. The dot at the top represents a nearby monument, the Albert Memorial statue. The central via an buildings are the college. For the purposes of this analogy, let the area mapped be our data. Below the map are three possible descriptions for the location of the department. On the left are a set of walking directions: Start at the statue, follow the highlighted roads, and you will arrive at the department. Secondly, there are longitude and latitude co-ordinates. Finally, we have simply the name of the department. All three identify the department in the mapped area: the first by a relationship with something else, the second by some property of the current building that houses it, and the third by a unique characteristic. Because all tell us how to identify a smaller area within the map, we consider the department *sub-data* of the mapped area.

However, we do not consider all of these descriptions to be *addresses*. A good address is one that identifies the target, no matter the frame of reference. Notice that the college is sub-data of the mapped area, but the area itself is sub-data of the entire United Kingdom. The walking directions are therefore *not* good directions in general; their starting point is just "Albert Memorial Statue", and there are many such statues in the UK. They will take you to a different place when when starting at the wrong statue. The co-ordinates are also not a good address. Were the department to move, they would no longer allow one to find it, as they identify the building rather than the department itself. However, the name of the department will aways identify it. It may not tell you how to find it, but it will certainly allow you to determine you have reached the right place.

We will use addresses to refer to sub-data within structures, and therefore need addresses that identify not just single data, but entire sub-structures. Moreover, the data will be mutated by program commands. We therefore need addresses that will be *robust* under mutation. They must survive the creation, destruction, and update of the data they are addressing. Much like the "department name" address above, we cannot use addresses that are tied to some transient property of the data.

Structural addresses will give us these properties. Compression and decompression using unique *body addresses*, when added as instrumentation, ensures that no machine command will have knowledge of them. We can think of these labels as "temporary signs", in that when we want to identify some sub-data, we erect a structural address, which gives us both a handle for the sub-data, and its relationship with the super-data. When we no longer need the sub-data, the signs are torn down, and may be reused elsewhere. Consider the following version of the figure 3.13 map. It represents the same area as before, but using structural addressing and decompression, a region within it has been highlighted.



We have no knowledge of the building that stands there, not even its name. However, the decompression using a structural address means it is clearly deliniated within the data. As long as the address **x** is not removed or duplicated, we will be able to find the region just by looking for **x**. Addresses need not tell us specifically *how* to find sub-data, They are not necessarily *instructions*, but rather unique "tags" or "handles'. As instrumentation layed over values, we will be able to identify them as distinct elements inside many types of data.

To capture our informal description, we define *structural addressing algebras*. These extend some set of values with structural addressing. Structural addressing algebras form the basis of abstract heap construction. If Values are the machine values of some structured heap, and (Values, StructAddrs, Data, addrs, comp) is a structural addressing algebra on Values, then Data are the values stored in the associated abstract heaps. The addresses StructAddrs form the body addresses of the data. The definition of compression and decompression are given by comp. The addresses function addrs helps limit the system to sane choices of comp.

Definition 42 (Structural addressing algebra)**.** Given an arbitrary countable set of values Values, a **structural addressing algebra** on Values consists of a countably infinite set of **structural addresses** StructAddrs (definition 35), a countable set of **addressable data** Data, an **addresses** function addrs, and a **compression** function comp:

$$\text{Structural addressing algebra} = \left(\begin{array}{c} \textsc{StructAddrs}, \textsc{Data}, \\ \text{addrs} : \textsc{Data} \to \mathcal{P}(\textsc{StructAddrs}), \\ \text{comp} : \textsc{StructAddrs} \xrightarrow{\text{fin}} \textsc{Data} \to \textsc{Data} \rightharpoonup \textsc{Data} \end{array} \right)$$

where StructAddrs is ranged over by $\mathbf{x}, \mathbf{y}, \mathbf{z}, \cdots$, Data is ranged over by $d, d_1, \cdots, d_n$, and the instance $\text{comp}(\mathbf{x}, d_1, d_2)$ is written $d_1 \circledast_{\mathbf{x}} d_2$. The following properties hold:

1. *Value containment*: Values are addressable data, Values $\subseteq$ Data.
2. *Unaddressed values*: For all $v \in$ Values, $\text{addrs}(v) = \emptyset$
3. *Address properties*: For all $d_1, d_2 \in$ Data and $\mathbf{x} \in$ StructAddrs, if $d_1 \circledast_{\mathbf{x}} d_2$ is defined then:
   a) *Containment*: $\mathbf{x} \in \text{addrs}(d_1)$.
   b) *Non-overlap*: $\text{addrs}(d_1) \cap \text{addrs}(d_2) \subseteq \{\mathbf{x}\}$
   c) *Preservation*: $(\text{addrs}(d_1) \setminus \{\mathbf{x}\}) \cup \text{addrs}(d_2) = \text{addrs}(d_1 \circledast_{\mathbf{x}} d_2)$
4. *Identity*: For all $d \in$ Data and $\mathbf{x} \in$ StructAddrs, there exists some $d_{\mathbf{x}} \in$ Data such that $d_{\mathbf{x}} \circledast_{\mathbf{x}} d = d$.
5. *Arbitrary addresses*: For all $d_1 \in$ Data and $\mathbf{x}, \mathbf{y} \in$ StructAddrs where $\mathbf{x} \in \text{addrs}(d_1)$ and $\mathbf{y} \notin \text{addrs}(d_1)$ or $\mathbf{y} = \mathbf{x}$, there exists $d_{\mathbf{y}} \in$ Data such that $d_1 \circledast_{\mathbf{x}} d_{\mathbf{y}}$ is defined, and that for all $d_2 \in$ Data, if $d_1 \circledast_{\mathbf{x}} d_2$ is defined, then $d_1 \circledast_{\mathbf{x}} d_{\mathbf{y}} \circledcirc_{\mathbf{y}} d_2 = d_1 \circledast_{\mathbf{x}} d_2$.
6. *Compression left-cancellativity*: For all $d_1, d_2, d_3 \in$ Data and $\mathbf{x} \in$ StructAddrs, if $d_1 \circledast_{\mathbf{x}} d_2 = d_1 \circledast_{\mathbf{x}} d_3$, then $d_2 = d_3$.
7. *Compression quasi-associativity*: For all $d_1, d_2, d_3 \in$ Data and $\mathbf{x}, \mathbf{y} \in$ StructAddrs where $\mathbf{y} \in \text{addrs}(d_2)$ and either $\mathbf{y} \notin \text{addrs}(d_1)$ or $\mathbf{x} = \mathbf{y}$, $(d_1 \circledast_{\mathbf{x}} d_2) \circledcirc_{\mathbf{y}} d_3 = d_1 \circledast_{\mathbf{x}} (d_2 \circledcirc_{\mathbf{y}} d_3)$.
8. *Compression quasi-commutativity*: For all $d_1, d_2, d_3 \in$ Data and $\mathbf{x}, \mathbf{y} \in$ StructAddrs where $\mathbf{x} \notin \text{addrs}(d_3)$ and $\mathbf{y} \notin \text{addrs}(d_2)$, $(d_1 \circledast_{\mathbf{x}} d_2 \circledcirc_{\mathbf{y}}) d_3 = (d_1 \circledcirc_{\mathbf{y}} d_3) \circledast_{\mathbf{x}} d_2$.

where undefined terms are considered equal.

These algebras are a minor variant on the multi-holed context algebras of [21]. Specifi-

cally, we have no longer require a number of the identity properties previously used. We also require that the algebra be built upon some set of values, representing unaddressed data.

The following properties of structural addressing algebras are useful:

- **Body addresses are optional**: The value containment property allows us work with abstract heaps that contain complete values (that is, values not containing any body addresses). This will ensure that normal structured heaps are a subset of abstract heaps. The unaddressed values property ensures that the choice of addressing does not conflict with data already present in values.

- **We can always find an address on which to decompress**: The arbitrary addresses property states that, if data can be decompressed, it can be decompressed using infinitely many addresses. Consider the decompression $d_1 \circledast d_2$. We can find an infinite subset of $s \subset \textsc{StructAddrs}$ such that $\text{addrs}(d_1) \cap s = \emptyset$, as each data is finite, and there are infinitely many addresses. For any $\mathbf{y} \in s$, there must be some $d_{\mathbf{y}}$ where$d_1 \circledast d_{\mathbf{y}}$ and $d_1 \circledast \mathbf{y} \circledcirc d_2$ is defined and equal to $d_1 \circledast d_2$ by the arbitrary addressing property. This last equality ensures that the new address has not meaningfully altered the structure of the data, as no matter what data is eventually placed into the address $\mathbf{y}$, it will give the same outcome as it did for address $\mathbf{x}$.

  This will allow abstract allocation where, if it happens that we chose an address that was already in use, we can find another. This will be useful in constructing abstract heaps, as abstract allocation will always require a fresh address to allocate a new abstract heap cell.

- **Data can be extended**: The address preservation, quazi-associativity and quazi-commutativity properties ensure that incomplete data are *separate* and can be *extended*.

  Take some $d \in \textsc{Data}$ that we wish to work with. To extend it "downward" with further sub-data, we can find some d_d and $\mathbf{x}$ where $d \circledast d_d$ is defined. To extend it upward with further super-data, we can find some d_u and $\mathbf{y}$ where $d_u \circledcirc d$ is defined. Quazi-associativity means that we need not consider which extension we knew about first: $d_u \circledcirc (d \circledast d_d)$ and $(d_u \circledcirc d) \circledast d_d$ are always equal. Notice that it may not be defined even if both parts are individually defined: address preservation ensures that the contained addresses will be the same regardless of which compression occurs first, but it may be that d_u and d_d are incompatible (e.g. both contain the same node identifier in the tree case).

Quazi-commutativity is similar, but covers the case where we have more than one sub-data; e.g. $d \circledast_{\mathbf{x}} d_{d_1}$ and $d \circledast_{\mathbf{y}} d_{d_2}$ both being defined for some data, and $\mathbf{x} \neq \mathbf{y}$. In this case, $d \circledast_{\mathbf{x}} d_{d_1} \circledast_{\mathbf{y}} d_{d_2} = d \circledast_{\mathbf{y}} d_{d_2} \circledast_{\mathbf{x}} d_{d1}$. Notice that whichever compression occurs first, address preservation ensures that the other address will not be destroyed.

These two cases ensure that we are free to *extend* data by collapsing new data into body addresses. The order of these extensions is irrelevant, as long as we do not introduce ambiguous addressing. Moreover, extending data via one address does not remove an existing ability to extend it via another. We can manipulate these data *separately.*

Both the abstract lists and abstract trees we defined are structural addressing algebras. Both follow follow directly from the definitions (e.g. structural addresses in definition 35 and abstract lists 36), and from compression being defined as linear substitution (definition 38).

Lemma 4 (Structural addressing algebra for lists)**.** *Given lists* Lists *and abstract lists* AbsLists *(definition 26) and definition 36 respectively) and structural addresses* StructAddrs *(definition 35), the tuple* (Lists, AbsLists, StructAddrs, *addrs*, *comp*) *is a structural addressing algebra.*

Lemma 5 (Structural addressing algebra for trees)**.** *Given trees* Trees *and abstract trees* AbsTrees *(definition 29 and definition 39 respectively) and structural addresses* StructAddrs *(definition 35), the tuple* (Trees, AbsTrees, StructAddrs, *addrs*, *comp*) *is a structural addressing algebra.*

We now formalise abstract heaps using structural addressing algebras. These heaps build upon a choice of structured heaps, using a structural address algebra to give notions of compression to the values, and structural addresses to add abstract heap cells.

Abstract heaps are the instrumented heaps associated with a choice of structured heap. Throughout this section, we will assume some choice of the following:

Assumptions 1 (Objects for abstract heap construction)**.** Assume the following parameters:

- A choice of underlying imperative machine on which to build an abstract heap. This includes machine addresses MachineAddrs, values MachineVals and structured heaps StructHeaps (parameters 1 and 2).
- The structural addresses of definition 35.

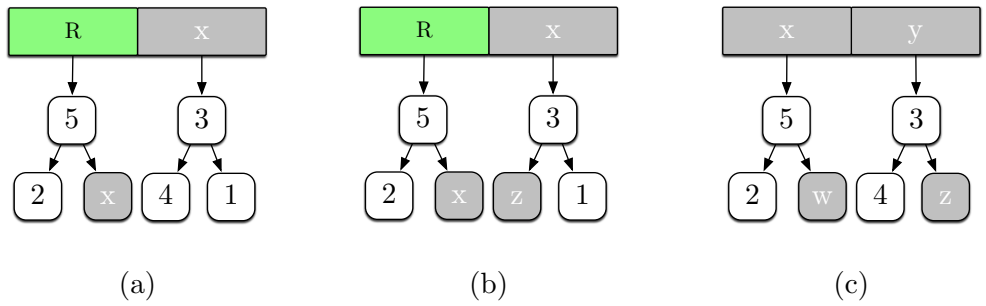


Figure 3.14.: One complete, and two incomplete abstract heaps. Heap (a) represents an abstract heap where all addresses used are present. Heap (b) has body address **x** matching with abstract heap address **x**. However, the body address **z** has no corresponding heap address. Heap (c) has body addresses **w** and **z** with no matching heap addresses. Moreover, the heap address **y** has no matching body address.

- A choice of structural addressing algebra (definition 42) on MachineVals using StructAddrs: (MachineVals, StructAddrs, Data, addrs, comp).

We build the definition of an abstract heap for the objects in assumptions 1. As abstract heaps are a superset of standard structured heaps, the set of instrumented addresses for them are the machine addresses plus a set of structural addresses.

Definition 43 (Abstract heap addresses)**.** Given the objects of assumptions 1, the set of **abstract heap addresses** Addrs, ranged over by $\mathbf{a}, \mathbf{a}_1, \cdots, \mathbf{a}_n$, is defined as:

$$\textsc{Addrs} \triangleq \textsc{MachineAddrs} \cup \textsc{StructAddrs}$$

The values stored in abstract heaps are drawn from the data of the structural algebra. Consider figure 3.14. Abstract heaps may be *complete*. Either they use just machine addresses and values or, as in case (a) of the figure, they are use structural addressing but data compress together to form machine values. Abstract heaps may also be *partial*, as in cases (b) and (c). These heaps consist of abstract heap cells with no matching body address or values containing body addresses that are not present in the heap domain.

We build the definition of abstract heaps in two stages. First, we define *pre-abstract heaps*, which are the naïve definition formed using the data of a structural addressing algebra as instrumented data. These heaps will have the undesirable properties we examined in the list case (figure 3.10). We test for these properties by defining the *collapse* process,

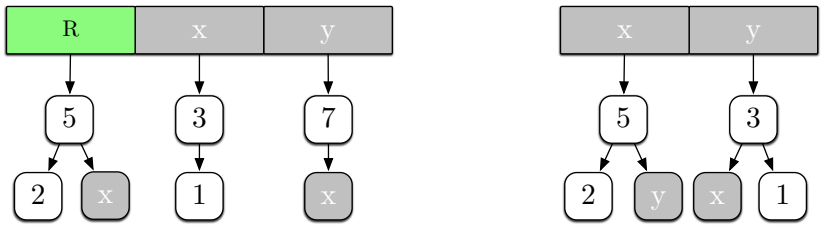


Figure 3.15.: On the left, an *ambiguous* pre-abstract heap, where the same address is used by two bodies. On the right, a *cyclic* pre-abstract heap, where structural addressing creates data that is connected to itself.

which compresses the structural addresses within abstract heaps (in essence, undoing abstract allocation). We then define the *abstract heaps* as these pre-abstract heaps which collapse to a concrete structured heaps.

Definition 44 (Pre-abstract heaps)**.** Given the objects of assumptions 1 and addresses ADDRS (definition 43), the set of **pre-abstract heaps** PREABSHEAPS, ranged over by $\mathbf{ph}, \mathbf{ph}_1, \cdots, \mathbf{ph}_n$, have the type:

$$\textsc{PreAbsHeaps} = \left\{ \mathbf{ph} \mid \mathbf{ph} : \textsc{Addrs} \xrightarrow{\text{fin}} \textsc{Data} \right\}$$

Pre-abstract heaps can be seen the union of three types of heap.

1. By construction, they contain all normal structured heaps (this follows from the value containment property of structural addressing algebras definition 42, and the addresses of definition 43).
2. They contain heaps with sensible uses of structural addressing. This means the values are either complete values, or structured data representing decompressed values that could be compressed together again.
3. They contain nonsensical uses of addressing, or use addressing to disguise badly formed data. These heaps describe no structured data at all. Their use of structural addresses may be *ambiguous*, where multiple values contain the same body address. They may also be *cyclic*, such that addressing connects data to itself. These cases are illustrated in figure 3.15.

The set of abstract heaps will be defined as those pre-abstract heaps with sensible addressing (points 1 and 2 above). We will filter out the nonsensical heaps via the *collapse*

operation. The *single-step collapse* relation describes collapsing one sub-data associated with some structural address in the heap domain into the matching body address of one super-data in the co-domain. It does this only if there is exactly one place in which the compression can occur (so that ambiguous addresses remain uncompressed).

Definition 45 (Single-step collapse relation)**.** Given the objects of assumptions 1, the **single-step collapse relation** $\downarrow \subset \textsc{PreAbsHeaps} \times \textsc{PreAbsHeaps}$ is defined such that, for all $\mathbf{ph}_1, \mathbf{ph}_2 \in \textsc{PreAbsHeaps}$, $(\mathbf{ph}_1, \mathbf{ph}_2) \in \downarrow$ if and only if both:

1. there exists some $\mathbf{x} \in \textsc{StructAddrs}$ and some unique $\mathbf{a} \in \textsc{Addrs}$ such that $\mathbf{x} \in \mathrm{dom}(\mathbf{ph}_1)$, $\mathbf{a} \in \mathrm{dom}(\mathbf{ph}_1)$, $\mathbf{a} \neq \mathbf{x}$, $\mathbf{x} \in \mathrm{addrs}(\mathbf{ph}_1(\mathbf{a}))$ and $\mathbf{ph}_1(\mathbf{a}) \circledast \mathbf{ph}_1(\mathbf{x})$ is defined;
2. the abstract cell at address $\mathbf{x}$ has been compressed into the matching body address, and removed from the heap; that is, $\mathbf{ph}_2 = \mathbf{ph}_1[\mathbf{a} \mapsto \mathbf{ph}_1(\mathbf{a}) \circledast \mathbf{ph}_1(\mathbf{x})]\downharpoonright_{\mathrm{dom}(\mathbf{ph}_1)\setminus\{\mathbf{x}\}}$.

An element $(\mathbf{ph}_1, \mathbf{ph}_2) \in \downarrow$ is written $\mathbf{ph}_1 \downarrow \mathbf{ph}_2$. The fact $\nexists \mathbf{ph}_2.\ \mathbf{ph}_1 \downarrow \mathbf{ph}_2$ is written $\mathbf{ph}_1 \not\downarrow$.

This single-step collapse removes exactly one structural address from a pre-abstract heap. The repeated use of the single-step collapse is expressed in the *collapse relation*.

Definition 46 (Collapse relation)**.** Given the objects of assumptions 1 and the single-step collapse relation of definition 45, the **collapse relation**, $\downarrow^* \subset \textsc{PreAbsHeaps} \times \textsc{PreAbsHeaps}$, is the reflexive, transitive closure of the single-step collapse relation $\downarrow$. If $(\mathbf{ph}_1, \mathbf{ph}_2) \in \downarrow^*$, we write $\mathbf{ph}_1 \downarrow^* \mathbf{ph}_2$.

The collapse relation can reveal invalid pre-abstract heaps, by reducing them to a form where no further collapses are possible. As this process essentially performs repeated abstract deallocation, if it "gets stuck" at any point, then the starting point cannot have been created via sensible uses of abstract allocation. For example, the heap $\mathrm{R} \mapsto 5[2 \otimes \mathbf{x}] * \mathbf{x} \mapsto 3[1] * \mathbf{y} \mapsto 7[\mathbf{x}]$ is stuck; it can collapse no further. The only body address ($\mathbf{x}$) is used ambiguously, and the single-step collapse relation will collapse only if there is a unique heap cell into which it can compress. Analogously, the heap $\mathbf{x} \mapsto 5[2 \otimes \mathbf{y}] * \mathbf{y} \mapsto 3[\mathbf{x} \otimes 1]$ is cyclic, and has two collapses that go no further: $\mathbf{x} \mapsto 5[2 \otimes 3[\mathbf{x} \otimes 1]]$ or $\mathbf{y} \mapsto 3[5[2 \otimes \mathbf{y}] \otimes 1]$.

Take any pre-abstract heap $\mathbf{ph}$ and machine heap h. If $\mathbf{ph} \downarrow^* h$, then it must be that the structural addressing used in $\mathbf{ph}$ was consistent, as every address could be compressed

away until only complete data remained. If this is possible, then h is the *only* machine heap that the collapse process can generate from $\mathbf{ph}$. We give the details of the proof in the appendix, section A.2. However, the result is easy to state.

Theorem 3 (Unique collapse)**.** *For all* $\mathbf{ph}_1 \in \textsc{PreAbsHeaps}$*, if there exists* $h \in \textsc{StructHeaps}$ *such that* $\mathbf{ph}_1 \downarrow^* h$*, then for all* $\mathbf{ph}_2 \in \textsc{PreAbsHeaps}$*,* $\mathbf{ph}_1 \downarrow^* \mathbf{ph}_2$ *and* $\mathbf{ph}_2 \not\downarrow$ *implies* $\mathbf{ph}_2 = h$*.*

This theorem is about pre-abstract heaps that are *complete*, in that they collapse to a structured heap. However, as stated, we will also work with *partial* pre-abstract heaps (figure 3.14). These heaps are critical, as they will represent abstract heaps where some of the data has been "framed off', and so will form the models of our local reasoning. We must therefore determine which pre-abstract heaps represent partial abstract heaps, a goal we achieve with *completion*. Completion adds in arbitrary extra resource to the domain of a pre-abstract heap, then determines if the result can collapse to a structured heap. If this was possible, then the partial heap must represent a portion of at least one useful abstract heap. If there is *no* additional resource we can add to which allows a pre-abstract heap to collapse to a structured heap, then it must be nonsense.

The completion process must consider adding both abstract heap cells and machine addressed heap cells. Recall figure 3.14. The heap labelled (b) will require an abstract heap cell addressed by $\mathbf{z}$ to complete the tree structure. The heap labelled (c) will require three heap cells to complete the tree structure: abstract cells $\mathbf{w}$ and $\mathbf{z}$, plus a machine cell at R.

Definition 47 (Pre-abstract heap completions)**.** Given the objects of assumption 1 and pre-abstract heaps $\textsc{PreAbsHeaps}$ (definition 44), the **completions** of a pre-abstract heap, $(\!|\cdot|\!) : \textsc{PreAbsHeaps} \to \mathcal{P}(\textsc{StructHeaps})$, are defined as:

$$(\!|\mathbf{ph}_1|\!) \triangleq \left\{ \; s \in \textsc{StructHeaps} \;\middle|\; \mathbf{ph}_2 \in \textsc{PreAbsHeaps}, (\mathbf{ph}_1 \sqcup \mathbf{ph}_2) \downarrow^* h \; \right\}$$

A pre-abstract heap $\mathbf{ph}$ is called **completable** if $(\!|\mathbf{ph}|\!) \neq \emptyset$.

We define the abstract heaps as all those pre-abstract heaps which have completions.

Definition 48 (Abstract heaps)**.** Given the objects of assumption 1, the **abstract heaps** $\textsc{AbsHeaps}$, ranged over by $\mathbf{h}, \mathbf{h}_1, \cdots, \mathbf{h}_n$, are those pre-abstract heaps which are completable.

$$\textsc{AbsHeaps} \triangleq \{\mathbf{ph} \in \textsc{PreAbsHeaps} \mid (\!|\mathbf{ph}|\!) \neq \emptyset\}$$

Both ambiguous and cyclic pre-abstract heaps have no completions, and are thus not abstract heaps. Recall figure 3.15. The ambiguous left-hand heap can never collapse to a machine heap, as the collapse process will never be able to completely remove the abstract cell $\mathbf{x}$ because it exists twice as a body address. The right-hand heap will never complete, as the collapse process will eventually reduce it to either $\mathbf{x} \mapsto 5[2 \otimes 3[\mathbf{x} \otimes 1]]$ or $\mathbf{y} \mapsto 3[5[2 \otimes \mathbf{y}] \otimes 1]$. It will collapse no further, as to get rid of, e.g., body address $\mathbf{x}$, it would have to have an abstract heap cell addressed $\mathbf{x}$, which is not possible as $\mathbf{x}$ already exists.

Lemma 6 (Abstract heaps are unambiguous)**.** *All* $\mathbf{h} \in \textsc{AbsHeaps}$ *are unambiguous, in that there are no* $\mathbf{x} \in \textsc{StructAddrs}$ *such that there exists* $\mathbf{a}_1, \mathbf{a}_2 \in \textsc{Addrs}$ *with* $\mathbf{x} \in \mathbf{h}(\mathbf{a}_1)$ *and* $\mathbf{x} \in \mathbf{h}(\mathbf{a}_2)$*.*

Lemma 7 (Abstract heaps are acyclic)**.** *All* $\mathbf{h} \in \textsc{AbsHeaps}$ *are acyclic, in that there are no structural addresses* $\mathbf{x} \in \textsc{StructAddrs}$ *and (possibly empty) chain of single-step collapse steps such that* $\mathbf{h} \downarrow^* \mathbf{h}_n$ *with* $\mathbf{x} \in \mathit{addrs}(\mathbf{h}_n(\mathbf{x}))$*.*

The proofs of these lemmas are in in the appendix, lemmas 28 and 29. Therefore, abstract heaps are either: complete machine heaps using no structural addressing; abstractly addressed heaps that collapse to complete machine heaps, so the addressing is merely a re-expression of the concrete data; or heaps in which structural addresses are present, but some super- or sub-data is not present. We can see any of these situations as describing a heap which is *consistent*. It may be that the heap is incomplete, but there must exist at least one extension to it that would result in a concrete structured heap.

3.3. Reasoning about abstract heaps

Reasoning with abstract heaps follows the pattern introduced in chapter 2. We first show that abstract heaps are instrumented structured heaps (parameter 8), and have primitive reification and composition compatible with the notion of views (definition 8). This ensures that our framework can be used with abstract heaps, and so provides us with a sound Hoare logic for program verification.

We select $\textsc{InstHeaps} = \textsc{AbsHeaps}$ in our framework. As the completion function on abstract heaps removes all structural addressing, generating the set of heaps *consistent* with the addresses, we pick the it to act as primitive reification.

Definition 49 (Primitive reification for abstract heaps)**.** Given the completion function $(\!|\cdot|\!)$ (definition 47), the **primitive reification function for abstract heaps**, $[\![:]\!]\textsc{AbsHeaps} \to \mathcal{P}(\textsc{StructHeaps})$, is defined as:

$$[\![\mathbf{h}]\!] = (\!|\mathbf{h}|\!)$$

Primitive composition for abstract heaps is disjoint function union, if the usage of structural addresses remains sensible.

Definition 50 (Primitive composition for abstract heaps)**.** The **primitive composition function for abstract heaps**, $\bullet : \textsc{AbsHeaps} \to \textsc{AbsHeaps} \rightharpoonup \textsc{AbsHeaps}$, is defined as:

$$\mathbf{h}_1 \bullet \mathbf{h}_2 \triangleq \begin{cases} \mathbf{h}_1 \sqcup \mathbf{h}_2 & \text{if } \mathbf{h}_1 \sqcup \mathbf{h}_2 \in \textsc{AbsHeaps} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We must prove associativity, commutativity and unit properties for primitive heap composition. The following lemma is helpful in this.

Lemma 8 (Sub-heaps of an abstract heap are abstract heaps)**.** *Let* $\mathbf{h} \in \textsc{AbsHeaps}$*. Then, for all* $\mathbf{ph}_1, \mathbf{ph}_2 \in \textsc{PreAbsHeaps}$ *if* $\mathbf{h} = \mathbf{ph}_1 \bullet \mathbf{ph}_2$ *then* $\mathbf{ph}_1, \mathbf{ph}_2 \in \textsc{AbsHeaps}$*.*

Proof. By definition, a pre-abstract heap $\mathbf{ph} \in \textsc{PreAbsHeaps}$ is an abstract heap if we can find some $\mathbf{ph}'$ and $h \in \textsc{StructHeaps}$ such that $\mathbf{ph} \sqcup \mathbf{ph}' \downarrow^* h$.

Assume that $\mathbf{h} = \mathbf{ph}_1 \bullet \mathbf{ph}_2$, $\mathbf{ph}_1, \mathbf{ph}_2 \in \textsc{AbsHeaps}$. By definition, this means $\mathbf{h} = \mathbf{ph}_1 \sqcup \mathbf{ph}_2$. Moreover, as $\mathbf{h} \in \textsc{AbsHeaps}$, there exists $\mathbf{ph}_3 \in \textsc{PreAbsHeaps}$ and $h \in \textsc{StructHeaps}$ such that $\mathbf{h} \sqcup \mathbf{ph} \downarrow^* h$. Take $\mathbf{ph}_1$, and let $\mathbf{ph}'_1 = \mathbf{ph}_2 \sqcup \mathbf{ph}_3$. By associativity and commutativity of $\sqcup$, it follows that $\mathbf{ph}_1 \sqcup \mathbf{ph}'_1 \downarrow^* h$, ergo $\mathbf{ph}_1 \in \textsc{AbsHeaps}$. Similar reasoning generates $\mathbf{ph}_2 \in \textsc{AbsHeaps}$. □

Lemma 9 (Primitive composition associates and commutes)**.** *The primitive composition operator for abstract heaps* $\bullet$ *(definition 50), associates and commutes with unit* $\{\} \to \{\}$*.*

Proof. Let $\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3 \in \textsc{AbsHeaps}$. Then:

- **Associativity**: We must show $(\mathbf{h}_1 \bullet \mathbf{h}_2) \bullet \mathbf{h}_3 = \mathbf{h}_1 \bullet (\mathbf{h}_2 \bullet \mathbf{h}_3)$. Assume $(\mathbf{h}_1 \bullet \mathbf{h}_2) \bullet \mathbf{h}_3$ is defined. Then, by lemma 8, the definition of $\bullet$ and the associativity of $\sqcup$, $(\mathbf{h}_2 \sqcup \mathbf{h}_3)$ is

a defined abstract heap. Theorem 3 and associativity of $\sqcup$ ensures that $\mathbf{h}_1 \sqcup (\mathbf{h}_2 \sqcup \mathbf{h}_3)$ is defined, and equal to $(\mathbf{h}_1 \bullet \mathbf{h}_2) \bullet \mathbf{h}_3$. The result follows.

- **Commutativity**: We must show $\mathbf{h}_1 \bullet \mathbf{h}_2 = \mathbf{h}_2 \bullet \mathbf{h}_1$. This follows directly from the definition of $\bullet$ as disjoin function union, and the commutativity thereof.
- **Unit**: We must show $\mathbf{h}_1 \bullet \{\} \to \{\} = \mathbf{h}_1$. This follows directly, as $\mathbf{h}_1 \bullet \{\} \to \{\} = \mathbf{h}_1 \sqcup \{\} \to \{\} = \mathbf{h}_1$.

□

Therefore, abstract heaps with the above notions of reification and composition is an addressed value view (definition 8). This, along with the results of our framework (chapter 2) ensures a sound local Hoare reasoning system.

Definition 51 (Abstract heaps build addressed value views)**.** Taking the set of instrumented heaps as AbsHeaps, primitive reification as $(\!|\cdot|\!)$ (definition 49), primitive composition as $\bullet$ (definition 50 and the identity as the identity function $\{\} \to \{\}$, the following addressed value view is generated:

$$(\mathcal{P}(\textsc{AbsHeaps}), \lfloor\cdot\rfloor, *, 0)$$

Reification

We have selected the completion function (definition 47) as our reification operation. This is a natural choice, as it means partial heaps are interpreted as *all* the machine heaps which they could plausibly represent. However, this choice does result in intuitionistic reasoning, similar to example 11. This is because the completions provide *arbitrary* additional resource, rather than just the resource needed to account for partial structured data. We can define an *extensional* notion of completions as an alternative, which adds only heap cells needed to account for missing structural addresses.

Definition 52 (Extensional Pre-abstract heap completions)**.** Given the objects of assumption 1, the **extensional completion** of a pre-abstract heap $(\!|\cdot|\!) : \textsc{PreAbsHeaps} \to \mathcal{P}(\textsc{StructHeaps})$ is defined as:

$$(\!|\mathbf{ph}|\!) \triangleq \left\{ s \in \textsc{StructHeaps} \;\middle|\; \begin{array}{l} \mathbf{ph}' \in \textsc{PreAbsHeaps}, (\mathbf{ph} \sqcup \mathbf{ph}') \downarrow^* s, \\ \forall \mathbf{a} \in \mathrm{dom}(\mathbf{ph}').\ \mathbf{a} \in \textsc{StructAddrs} \\ \quad \implies \exists \mathbf{b} \in \mathrm{dom}(\mathbf{ph}).\ \ \mathbf{a} \in \mathrm{addrs}(\mathbf{ph}(\mathbf{b})), \\ \nexists \mathbf{a} \in \mathrm{dom}(\mathbf{ph}').\ \mathbf{ph}(\mathbf{a}) = s(\mathbf{a}) \end{array} \right\}$$

There are two new conditions on this when compared to definition 47. The first states that new abstract heap cells are added *only* to account for body addresses that had no abstract heap cell with a matching structural address. The second ensures that no heap cells are added that remain unchanged during the collapse process (and so were not needed for collapsing). Therefore, the only data added is that which enables a previously impossible collapse step.

We call this different completion *extensional* as it extends the structured data to account for partial data. We call the choice of definition 49 *intuitionistic* to match the previous separation logic terminology. In this thesis, we use intuitionistic completions unless otherwise stated.

Comment 5. The intuitionistic approach extends data to complete it, and extends the heap with arbitrary cells. The extension approach is designed to *only* extend data, but can still generate new heap cells to ensure that all abstract heap cells have a machine heap cell into which they can collapse. I choose to use the intuitionistic approach, as the extensional definition seems rather a "half way house". It can be used to provide *arbitrary* additional structured data with, for example, heaps such as $\mathbf{x} \mapsto \varnothing$ (an abstract list heap consisting of one abstract cell with no contents), but provides at most one machine heap cell. I do not find it useful to allow one and not the other, and given that we *must* provide some level of completions to give meaning to partial abstract heaps, working with the intuitionistic case seems the most natural choice.

3.3.1. Data assertions

That abstract heaps form an addressed value algebra (definition 51) ensures our framework provides a sound Hoare logic using abstract heaps as models. We need only create an assertion language to describe them. We use two types of syntactic assertion. The first type, **abstract data assertions**, describes abstractly addressed data, and will largely be

defined on a model-by-model basis. These assertions describe the *contents* of structured and abstract heap cells, and there will be different assertion languages for lists, for trees, and for the DOM and POSIX examples of later chapters. However, all these languages must support a set of common operations, including *compression*.

The second type, **abstract heap assertions**, will be common to all libraries, and describes abstract heaps themselves. These assertions are parameterised on the data assertions. The abstract heap assertions describe entire abstract heaps, including the cells and separation between them.

Abstract data assertions

The set of assertions describing structurally addressed data consists of assertions useful for all types of data, and those chosen for specific data models.

Definition 53 (Assertions about arbitrary structurally addressed data)**.** Given a set of logical variables LVars (parameter 13) and logical expressions LExprs (parameter 14), the set of **data assertions** DataAssts, ranged over by $\phi, \phi_1, \cdots, \phi_n$, is defined by induction as: for all $\alpha, \mathrm{x} \in$ LVars, $\mathrm{E} \in$ LExprs

$$
\begin{array}{rll}
\phi ::= & \phi_1 \wedge \phi_2 & \text{Conjunction} \\
| & \phi_1 \vee \phi_2 & \text{Disjunction} \\
| & \phi_1 \Rightarrow \phi_2 & \text{Implication} \\
| & \mathbf{false} & \text{Falsity} \\
| & \exists \mathrm{x}.\ \phi & \text{Existential logical variable} \\
| & \mathrm{E} & \text{Logical expression} \\
| & \phi_1 \circledcirc \phi_2 & \text{Address compression} \\
| & \diamond \alpha & \text{Address availability} \\
| & \psi & \text{Data-model specific assertions}
\end{array}
$$

where the data-model specific assertions are defined on a data model by data model basis.

The majority of these assertions are standard. Of note are *address compression* and *address availability*. Address compression $\phi_1 \circledcirc \phi_2$ describes data created by compressing sub-data ϕ_2 into super-data ϕ_1 on the body address contained in logical variable α. This is a binding operation, so that α is *bound* within ϕ_1. Recalling that decompression is defined in terms of compression, address compression also allows us to describe data which can be *decompressed*. The assertion will be used to describe situations where ϕ_2 can be abstractly

allocated. The address availability assertion $\diamond\alpha$ describes data containing body address α. Ergo, it describes *incomplete* data; data in which *some* sub-data can be compressed. This will be used to describe abstract deallocation.

The data assertions are interpreted with logical environments into sets of data.

Definition 54 (Data assertion interpretation)**.** Given logical expressions with evaluation function $(\!|\cdot|\!)(\Gamma)$ (parameter 14) on logical environments LEnvs (definition 20), data assertions DataAssts (definition 57), and data Data of some structural addressing algebra (definition 42), the assertion interpretation function $(\!|\cdot|\!)^{\cdot} : \text{LEnvs} \to \text{DataAssts} \to \mathcal{P}(\text{Data})$ must satisfy: for all $\phi_1, \phi_2 \in \text{DataAssts}, \Gamma \in \text{LEnvs}$

$$
\begin{aligned}
(\!|\phi_1 \wedge \phi_2|\!)^{\Gamma} &\triangleq (\!|\phi_1|\!)^{\Gamma} \cap (\!|\phi_2|\!)^{\Gamma} \\
(\!|\phi_1 \vee \phi_2|\!)^{\Gamma} &\triangleq (\!|\phi_1|\!)^{\Gamma} \cup (\!|\phi_2|\!)^{\Gamma} \\
(\!|\phi_1 \Rightarrow \phi_2|\!)^{\Gamma} &\triangleq (\text{Data} \setminus (\!|\phi_1|\!)^{\Gamma}) \cup (\!|\phi_2|\!)^{\Gamma} \\
(\!|\mathbf{false}|\!)^{\Gamma} &\triangleq \emptyset \\
(\!|\exists \mathrm{x}.\ \phi|\!)^{\Gamma} &\triangleq \bigcup_{x \in \text{LVals}} (\!|\phi|\!)^{\Gamma[\mathrm{x} \mapsto x]} \\
(\!|\mathrm{E}|\!)^{\Gamma} &\triangleq \langle\!|\mathrm{E}|\!\rangle(\Gamma) \\
(\!|\phi_1 \mathbin{@} \phi_2|\!)^{\Gamma} &\triangleq \{d_1 \circledast d_2 \mid d_1 \in (\!|\phi_1|\!)^{\Gamma}, d_2 \in (\!|\phi_2|\!)^{\Gamma}, \mathbf{x} = \Gamma(\alpha)\} \\
(\!|\diamond\alpha|\!)^{\Gamma} &\triangleq \{d \in \text{Data} \mid \Gamma(\alpha) \in \text{addrs}(d)\} \\
(\!|\psi|\!)^{\Gamma} &\triangleq \text{Defined on model-by-model basis}
\end{aligned}
$$

The majority of the interpretations are standard. As stated, address compression describes data that results can be de-compressed into two two sub-data on an address. Address availability describes data that will accept some sub-data into the specified structural address. The data model specific assertions will depend on the data structure being considered, and will typically be syntactic versions of the data structure algebra.

3.3.2. Abstract heap assertions

Recall the heap cell assertions used in our primitive heap example (example 17). We define a similar *abstract heap cell* assertion here, where the values of the cell are described using the data assertions of definition 57. We also include standard variable cells, and the assertion that describes expression evaluation.

Definition 55 (Abstract heap assertions)**.** Given program variables PVars (definition 3), logical expressions LExprs (parameter 14), addresses Addrs (definition

43), data assertions DataAssts (definition 57), the *abstract heap cell* assertions are defined as: for all $\mathbf{x} \in \text{PVars}, \mathrm{E} \in \text{LExprs}, \mathbf{a} \in \text{Addrs}, \phi \in \text{DataAssts}, e \in \text{Exprs}$

$$
\begin{array}{rrr}
P ::= & \mathbf{x} \to \mathrm{E} & \text{Variable cell} \\
| & \mathrm{E} \mapsto \phi & \text{Heap cell} \\
| & e \Rightarrow \mathrm{E} & \text{Expression evaluation}
\end{array}
$$

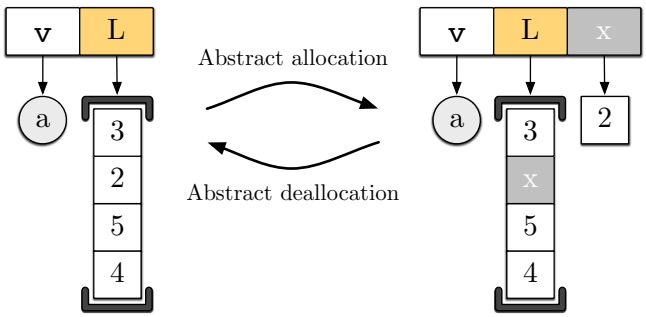
Given a logical expression evaluation function $(\!|\cdot|\!)(\Gamma)$, (parameter 14), the assertion interpretations are:

$$
\begin{array}{rcl}
(\!|\mathbf{x} \to \mathrm{E}|\!)^{\Gamma} & \triangleq & \{\{\mathbf{x} \to V\} \mid V \in (\!|\mathrm{E}|\!)(\Gamma)\} \\
(\!|\mathrm{E} \mapsto \psi|\!)^{\Gamma} & \triangleq & \{\{\mathbf{a} \mapsto d\} \mid \mathbf{a} = (\!|\mathrm{E}|\!)(\Gamma), d \in (\!|\psi|\!)^{\Gamma}\} \\
(\!|e \Rightarrow \mathrm{E}|\!)^{\Gamma} & \triangleq & \{v \in \text{Views} \mid \forall s \in \lfloor v \rfloor.\ (\![e]\!)(v) = (\!|\mathrm{E}|\!)(\Gamma)\} \cap \text{safe}(()e)
\end{array}
$$

Notice that, as Addrs contains both normal and structural addresses, we need only one assertion to describe both standard heap cells and abstractly addressed cells. Abstract heap assertions allow us to describe entire abstract heaps.

3.3.3. Abstract allocation

We have now given the constructions of abstract heaps, shown how they form models of our reasoning framework (chapter 2), and created a general assertion language for them. We now demonstrate how abstract heap cells are created and destroyed in the reasoning process. The key observation is that two abstract heaps differing only by uses of abstract allocation and deallocation of some data are *identical* under reification on abstract heaps. Recalling figure 3.1 (the diagram of which is reproduced below for convenience), both sides of the allocation have identical reifications, as $\downarrow$ collapses $\mathbf{x}$ and produces the original list (definition 49).



Recall the semantic consequence relation defined in 17. It allows the instrumentation associated with heaps to be altered in proofs, as long as their reifications are not meaningfully changed under any frames. This is precisely the property we require for abstract allocation, if we can be assured that allocating a new abstract heap cell does not clash with any others. We can guarantee this with the standard technique of existentially quantifying over possible addresses (which we have previously used for primitive heap allocation in example 6 and tree node allocation in definition 62). The following *semantic equivalence*, which we call the *abstract allocation and deallocation* property, is therefore justified:

Theorem 4 (Abstract allocation and deallocation)**.** *Given the semantic consequence relation* $\preccurlyeq_\Gamma$ *(definition 23), data assertions* DataAssts *(definition 57), abstract heap assertions* Assts *(definition 55) and logical variables* LVars *(parameter 13), the following semantic equivalence is valid: for all* $\alpha, \beta \in$ LVars*,* $\phi_1, \phi_2 \in$ DataAssts*, and* γ *is not free in* ϕ_1 *or* ϕ_2

$$(\alpha \mapsto \phi_1 \circledast_\beta \phi_2) \preccurlyeq\succcurlyeq_\Gamma \exists\gamma.\, (\alpha \mapsto (\phi_1 \circledast_\beta \gamma \wedge \diamond\gamma) * \gamma \mapsto \phi_2)$$

This is the natural property we want for the reasoning. Whenever a proof requires granular data access, we can allocate an abstract heap cell for any addressable sub-data. Whenever the proof no longer needs such fine access, we can collapse the sub-data away.

Reading the equivalence left-to-right, it states "If we can find some heap cell at address α with data that can, via some address, be decompressed into ϕ_1 and ϕ_2, then we can perform the decompression and allocate a cell γ containing ϕ_2 whilst knowing for certain that it will compose with the ϕ_1 left behind in cell α". The converse reading is also valid, stating that if we have two heap cells that can be compressed, we can deallocate the abstract cell. Notice the use of existential quantification on the address being allocated. This is critical, as this re-expression of the heap must not invalidate any frames. When deallocating, we do not need the existential quantifier, as the address is bound by the compression operation. We use a fresh logical variable γ to capture the fresh address, as it is possible that ϕ_2 contains an unbound instance of β. Context application ensures that γ takes the place of β in ϕ_1.

Proof. (Theorem 4). First, unroll the equivalence via the consequence assertion to obtain the following pair of semantic consequences, equivalent to the proposition. The left to right case is:

$$\forall\Gamma \in \text{LEnvs}.\, (\llparenthesis(\alpha \mapsto (\phi_1 \circledast_\beta \phi_2))\rrparenthesis^\Gamma \preccurlyeq \llparenthesis\exists\gamma.\ (\alpha \mapsto (\phi_1 \circledast_\beta \gamma \wedge \diamond\beta) * \gamma \mapsto \phi_2)\rrparenthesis^\Gamma)$$

The right to left case is:

$$\forall \Gamma \in \text{LEnvs}.\ \left(\!\left|\exists \gamma.\ \left(\alpha \mapsto (\phi_1 \mathbin{\textcircled{\scriptsize β}} \gamma \wedge \diamond\beta) * \gamma \mapsto \phi_2\right)\right|\!\right)^{\Gamma} \preccurlyeq \left(\!\left|\left(\alpha \mapsto (\phi_1 \mathbin{\textcircled{\scriptsize β}} \phi_2)\right)\right|\!\right)^{\Gamma}\right)$$

To show the proposition, we must show that both of these semantic consequences hold. Fix some arbitrary $\Gamma \in \text{LEnvs}$, $\mathbf{x} = \Gamma(\alpha), \mathbf{y} = \Gamma(\beta), \mathbf{y} = \Gamma(\gamma)$. By further unpacking the definition of assertion interpretation and semantic consequence, for the left to right case, we must show: for all $r \in \text{Views}$

$$\begin{array}{c} \left\lfloor \left\{ \mathbf{x} \mapsto (d_1 \mathbin{\textcircled{\scriptsize y}} d_2) \bullet r \mid d_1 \in (\!|\phi_1|\!)^{\Gamma[\beta \mapsto \mathbf{y}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\beta \mapsto \mathbf{y}]} \right\} \right\rfloor \\ \subseteq \\ \left\lfloor \bigvee_{\mathbf{y} \in \text{StructAddrs}} \cdot \left\{ \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \mid d_1 \in (\!|\phi_1 \mathbin{\textcircled{\scriptsize β}} \gamma \wedge \diamond\gamma|\!)^{\Gamma[\beta \mapsto \mathbf{y}, \gamma \mapsto \mathbf{z}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\gamma \mapsto \mathbf{z}]} \right\} \right\rfloor \end{array}$$

For the right to left case, we must show the converse of the above. Hence, we are actually showing *equality* between the two sets. By unpacking the definition of reification and disjunction, we obtain the following for the left hand side:

$$\begin{array}{rl} & \left\lfloor \left\{ \mathbf{x} \mapsto (d_1 \mathbin{\textcircled{\scriptsize y}} d_2) \bullet r \mid d_1 \in (\!|\phi_1|\!)^{\Gamma[\beta \mapsto \mathbf{y}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\beta \mapsto \mathbf{y}]} \right\} \right\rfloor \\ = & \left\{ \left\lfloor\!\left\lfloor \mathbf{x} \mapsto (d_1 \mathbin{\textcircled{\scriptsize y}} d_2) \bullet r \right\rfloor\!\right\rfloor \mid d_1 \in (\!|\phi_1|\!)^{\Gamma[\beta \mapsto \mathbf{y}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\beta \mapsto \mathbf{y}]} \right\} \end{array}$$

and the following for the right:

$$\begin{array}{rl} & \left\lfloor \bigvee_{\mathbf{z} \in \text{StructAddrs}} \cdot \left\{ \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \mid d_1 \in (\!|\phi_1 \mathbin{\textcircled{\scriptsize β}} \gamma \wedge \diamond\gamma|\!)^{\Gamma[\beta \mapsto \mathbf{y}, \gamma \mapsto \mathbf{z}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\gamma \mapsto \mathbf{z}]} \right\} \right\rfloor \\ = & \left\lfloor \bigcup_{\mathbf{z} \in \text{StructAddrs}} \cdot \left\{ \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \mid d_1 \in (\!|\phi_1 \mathbin{\textcircled{\scriptsize β}} \gamma \wedge \diamond\gamma|\!)^{\Gamma[\beta \mapsto \mathbf{y}, \gamma \mapsto \mathbf{z}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\beta \mapsto \mathbf{z}]} \right\} \right\rfloor \\ = & \bigcup_{\mathbf{z} \in \text{StructAddrs}} \cdot \left\{ \left\lfloor\!\left\lfloor \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \right\rfloor\!\right\rfloor \mid d_1 \in (\!|\phi_1 \mathbin{\textcircled{\scriptsize β}} \gamma \wedge \diamond\gamma|\!)^{\Gamma[\beta \mapsto \mathbf{y}, \gamma \mapsto \mathbf{z}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\beta \mapsto \mathbf{z}]} \right\} \end{array}$$

The result follows if these two are sets equal for any choice of r. That is, we must prove:

$$\begin{array}{c} \left\{ \left\lfloor\!\left\lfloor \mathbf{x} \mapsto (d_1 \mathbin{\textcircled{\scriptsize y}} d_2) \bullet r \right\rfloor\!\right\rfloor \mid d_1 \in (\!|\phi_1|\!)^{\Gamma[\beta \mapsto \mathbf{y}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\beta \mapsto \mathbf{y}]} \right\} \\ = \\ \bigcup_{\mathbf{z} \in \text{StructAddrs}} \cdot \left\{ \left\lfloor\!\left\lfloor \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \right\rfloor\!\right\rfloor \mid d_1 \in (\!|\phi_1 \mathbin{\textcircled{\scriptsize β}} \gamma \wedge \diamond\gamma|\!)^{\Gamma[\beta \mapsto \mathbf{y}, \gamma \mapsto \mathbf{z}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\gamma \mapsto \mathbf{z}]} \right\} \end{array}$$

The easier case is right-to-left. Pick any element $\left\lfloor\!\left\lfloor \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \right\rfloor\!\right\rfloor$ from the right-hand side. Then:

$$\left\lfloor\!\left\lfloor \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \right\rfloor\!\right\rfloor = \left\lfloor\!\left\lfloor \mathbf{x} \mapsto (d_1 \mathbin{\textcircled{\scriptsize z}} d_2) \bullet r \right\rfloor\!\right\rfloor$$

$$
\begin{array}{ll}
 & \mathrm{R} \mapsto 1[2 \otimes 3] \\
= & \exists \alpha.\ ((\mathrm{R} \mapsto 1[2 \otimes \alpha]) \textcircled{@} 3) \\
\preccurlyeq_\Gamma & \exists \alpha.\ ((\mathrm{R} \mapsto 1[2 \otimes \alpha] \wedge \diamond\alpha) * \alpha \mapsto 3) \\
\preccurlyeq_\Gamma & \exists \alpha.\ (\mathrm{R} \mapsto 1[2 \otimes \alpha] * \alpha \mapsto 3)
\end{array}
$$

Figure 3.16.: Example of abstract allocation splitting a tree heap (definition 39) with three nodes by abstractly allocating a new heap cell α containing the 3 node.

This follows by the definition of reification as completions (defined in terms of compression), the confluence of completion, the fact that $\mathbf{z} \in \mathrm{addrs}(d_1)$ by $d_1 \in (\!|\phi_1 \textcircled{\beta} \gamma \wedge \diamond\gamma|\!)^{\Gamma[\beta \mapsto \mathbf{y}, \gamma \mapsto \mathbf{z}]}$, and the address irrelevence property that ensures $d_1 \textcircled{\mathbf{y}} \mathbf{z} \textcircled{\mathbf{z}} d_2 = d_1 \textcircled{\mathbf{y}} d_2$. Then, $[\![\mathbf{x} \mapsto (d_1 \textcircled{\mathbf{y}} d_2) \bullet r]\!]$ is in the left hand side by observation (note that the choice of $\mathbf{y}$ is irrelevant, as it is bound on the left).

Left-to-right is harder, as the introduction of the new abstract cell $\mathbf{z}$ could plausibly conflict with some cells in the frame r. To justify this, we use the same argument that supports allocation of normal heap cells primitive heaps (example 16). As every datum is finite, and r is a finite map, there are only finitely many "used" structural addresses. By the infinite addresses and address irrelevance properties of structural addressing algebras (definition 42), we can thus replace $\mathbf{y}$ with some alternative no matter what choice of r is made. So:

$$
\begin{array}{rl}
 & \{\lfloor \mathbf{x} \mapsto (d_1 \textcircled{\mathbf{y}} d_2) \bullet r \rfloor \mid d_1 \in (\!|\phi_1|\!)^{\Gamma[\beta \mapsto \mathbf{y}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\beta \mapsto \mathbf{y}]}\} \\
= & \{\lfloor \mathbf{x} \mapsto d_1 \bullet \mathbf{z} \mapsto d_2 \bullet r \rfloor \mid d_1 \in (\!|\phi_1 \textcircled{\beta} \gamma \wedge \diamond\gamma|\!)^{\Gamma[\beta \mapsto \mathbf{y}, \gamma \mapsto \mathbf{z}]}, d_2 \in (\!|\phi_2|\!)^{\Gamma[\gamma \mapsto \mathbf{z}]}\}
\end{array}
$$

for *some* choice of $\mathbf{z}$, noting that $\phi_1 \textcircled{\beta} \gamma \wedge \diamond\gamma$ is non-empty when ϕ_1 is non-empty by the arbitrary addresses property of structural addressing algebras. As, regardless of the choice of r, we can find some $\mathbf{z}$, the inclusion will holds as we consider *every* $\mathbf{z}$ in the union. □

As the abstract allocation relation is contained within semantic consequence, we can use the rule of semantic consequence to perform abstract allocation within a proof. For example, the derivation of figure 3.16 is justified by theorem 4.

Comment 6. I see the allocation of abstract heap cells as similar to the use of consequence for (un)rolling inductive predicates in separation logic (discussed in section 1.2.2). Just as an inductive predicate "unrolls" to allow access to the internals of the

structure, we can "allocate" the internals of some rich data. As inductive predicates roll up, we deallocate. Thanks to semantic consequence, the same proof rule justifies both styles of reasoning.

3.3.4. Axiomatic specifications for lists and trees

We now give data assertions and axioms for our list library (definition 26). To give these axioms, we will extend the standard logical values to include abstract lists, and logical expressions to include a list membership expression $\mathrm{E}_1 \in \mathrm{E}_2$ which tests if the list describe by E_1 is a sublist of the list described by E_2. This will be used in the `append` command (definition 58) to show that an element does not already exist in the list.

Definition 56 (Logical values and expressions for lists)**.** Given the set of addresses Addrs (definition 43) and abstract lists AbsLists (definition 36), the set of logical values for lists LVals is defined as:

$$\text{LVals} \triangleq \mathbb{Z} \cup \mathbb{B} \cup \text{Addrs} \cup \text{AbsLists}$$

Logical values are thus the standard variable types plus partial lists. The set of logical expressions and their interpretations are those provided by framework (parameter 14), along with *list membership*:

$$\mathrm{E}_1 \in \mathrm{E}_2$$

The interpretation of this expression, in terms of the interpretation of the standard expressions (parameter 14):

$$(\!|\mathrm{E}_1 \in \mathrm{E}_2|\!)(\Gamma) \triangleq \begin{cases} \textbf{true} & \text{if} (\!|\mathrm{E}_1|\!)(\Gamma) = i, (\!|\mathrm{E}_2|\!)(\Gamma) = \mathbf{l}_3, \mathbf{l}_3 = \exists \mathbf{l}_1, \mathbf{l}_2.\ \mathbf{l}_1 \otimes i \otimes \mathbf{l}_2 \\ \textbf{false} & \text{otherwise} \end{cases}$$

The assertions specific to lists are a simple lift of the abstract list data (definition 36).

Definition 57 (Assertions on lists)**.** Given a set of logical expressions LExprs and values LVals for lists (definition 56), the set of **list specific data assertions**, providing the model-specific choices for definition , are: for all $\mathrm{E} \in \text{LExprs}, \alpha \in \text{LVals}$

$$
\begin{array}{rll}
\psi ::= & \mathrm{E} & \text{Element/Logical expression} \\
| & \psi_1 \otimes \psi_2 & \text{List concatenation} \\
| & \varnothing & \text{Empty list} \\
| & [\psi] & \text{Complete list} \\
| & \alpha & \text{Body address}
\end{array}
$$

The data assertion interpretation function is extended by:

$$
\begin{array}{rcl}
(\!|\mathrm{E}|\!)^{\Gamma} & \triangleq & \{\!|\mathrm{E}|\!\}(\Gamma) \\
(\!|\psi_1 \otimes \psi_2|\!)^{\Gamma} & \triangleq & \{\mathbf{l}_1 \otimes \mathbf{l}_2 \mid \mathbf{l}_1 \in (\!|\psi_1|\!)^{\Gamma}, \mathbf{l}_2 \in (\!|\psi_2|\!)^{\Gamma}\} \\
(\!|\varnothing|\!)^{\Gamma} & \triangleq & \{\varnothing\} \\
(\!|[\psi]|\!)^{\Gamma} & \triangleq & \{[\mathbf{pl}] \mid \mathbf{pl} \in (\!|\psi|\!)^{\Gamma}\} \\
(\!|\alpha|\!)^{\Gamma} & \triangleq & \{\mathbf{x} \mid \mathbf{x} = \Gamma(\alpha)\}
\end{array}
$$

where, if the results are not sets of abstract lists, the interpretation is undefined.

These assertions allow us to give syntactic axioms to the list commands. Recall the abstract heaps for lists of definition 27, and the abstract heap assertions (definition 55).

Definition 58 (List update command axioms)**.** The **syntactic axioms for the commands of the list module** (given in definition 28) are as follows: for all $P \in$ Assts, P

$$
\begin{array}{ccc}
\left\{ \begin{array}{c} (\mathrm{L} \mapsto [\mathrm{A} \wedge \neg\exists\alpha.\ \diamond\alpha] * P) \\ \wedge\ e \Rightarrow \mathrm{I} \wedge \mathrm{I} \notin \mathrm{A} \end{array} \right\} & \texttt{append}(e) & \{\mathrm{L} \mapsto [\mathrm{A} \otimes \mathrm{I}] * P\} \\
\{\alpha \mapsto \mathrm{I} * P \wedge e \Rightarrow \mathrm{I}\} & \texttt{remove}(e) & \{\alpha \mapsto \varnothing * P\} \\
\{\texttt{i} \to \texttt{-} * \mathrm{L} \mapsto [\mathrm{I} \otimes \alpha]\} & \texttt{i := getFirst()} & \{\texttt{i} \to \mathrm{I} * \mathrm{L} \mapsto [\mathrm{I} \otimes \alpha]\} \\
\{\texttt{i} \to \texttt{-} * \mathrm{L} \mapsto [\varnothing]\} & \texttt{i := getFirst()} & \{\texttt{i} \to 0 * \mathrm{L} \mapsto [\varnothing]\} \\
\{(\texttt{j} \to \texttt{-} * \alpha \mapsto \mathrm{I} \otimes \mathrm{J} * P) \wedge e \Rightarrow \mathrm{I}\} & \texttt{j := getRight}(e) & \{\texttt{j} \to \mathrm{J} * \alpha \mapsto \mathrm{I} \otimes \mathrm{J} * P\} \\
\{(\texttt{j} \to \texttt{-} * \mathrm{L} \mapsto [\alpha \otimes \mathrm{I}] * P) \wedge e \Rightarrow \mathrm{I}\} & \texttt{j := getRight}(e) & \{\texttt{j} \to 0 * \mathrm{L} \mapsto [\alpha \otimes \mathrm{I}] * P\}
\end{array}
$$

The semantic axioms are these syntactic axioms interpreted through all logical environments.

The axiom pre-conditions reflect the resource required to ensure fault freedom. Consider the pre-condition for `append(i)`, given by: $(\mathrm{L} \mapsto [\mathrm{A} \wedge \neg\exists\alpha.\ \diamond\alpha] * P) \wedge e \Rightarrow \mathrm{I} \wedge \mathrm{I} \notin \mathrm{A}$. It demands the resource needed to evaluate e, but also the *entire* list. This is because the command will fault if the result of evaluating $e$is already in the list. The only way

to ensure that the command does not fault is to analyse every element of the list. We can capture this using the complete list assertion $[\text{A}]$ with logical variable A, but must additionally ensure that no data has been abstractly allocated (as the element could then be in another abstract heap cell). This is enforced by the assertion $\neg\exists\alpha.\ \diamond\alpha$, which states that no structural addresses exist in the value of A. This idiom occurs frequently enough for us to introduce a predicate for it:

Definition 59 (Complete data predicate)**.** The **is_complete** predicate is defined as:

$$\mathit{is\_complete} \triangleq \neg\exists\alpha.\ \diamond\alpha$$

The `remove` command requires only the element to be removed, so the pre-condition assertion $\alpha \mapsto \text{I}$ only needs to refer to the partial list with just element I. As e is a program expression, it must evaluate to a natural number in this situation, so the abstract cell must contain a single element. This axiom demonstrates the advantage of the "abstract allocation" approach. The resulting pre-condition is concise, and clearly defines the behavior of the command. Similarly, `getFirst` is well described. If there is a first element, it requires only the evidence of that element (the I), and the rest of the list can be removed by allocation using α. If there is no first element, it requires the empty list as evidence. These axioms exactly describe the actions given to the operation semantics in definition 28, but by virtual of being written in a local manner, we claim are a more intuitive description of the command behaviours.

Soundness of the commands is shown as in the examples of chapter 2. Notice however that some list commands have multiple axioms. This is expected, as richer libraries have commands with state dependent behaviours. In these cases, it is important that at most one of the pre-conditions holds at any one time. If multiple pre-conditions were applicable, the choice of which to use would fall to the person constructing the proof. This results in situations where the program must behave as the *proof* wanted when multiple options are available, rather than the proof behave as the program requires.

Lemma 10 (Atomic soundness of list commands)**.** *Given structural addressing algebra on abstract lists* $(\text{VIEWS}, \lfloor\cdot\rfloor, *, 0)$ *(definitions 42 and 36 respectively), the list commands are atomically sound with respect to their axioms (definition 58), thus satisfying the atomic soundness requirement of the framework (definition 12).*

Proof. Atomic soundness requires that, for each semantic axiom $\{p\}\ \text{C}\ \{q\}$, $\forall r \in \text{VIEWS}.\ [\![\text{C}]\!]\lfloor p * r\rfloor \subseteq \lfloor q * r\rfloor$. We show soundness of the first three commands

(`getRight` is similar).

1. `append(i)`: For any frame $r \in$ VIEWS, interpretation and reification of the pre-condition results in sets of heaps with the form $\{\mathrm{L} \mapsto [pl]\} \sqcup h$ where h contains enough resource to evaluate e, and where no frame nor completion can affect the contents of pl as it must contain no structural addresses. The translation of the conjunct $\mathrm{I} \notin \mathrm{L}$ ensures that $i \notin pl$. Therefore, the action of `append` on these heaps is to add a new element i to the end of the list, resulting in heaps of the form $\{\mathtt{i} \mapsto i, \mathrm{L} \mapsto [pl \otimes i]\} \sqcup h$. This is exactly the translation and reification of the post-condition.
2. `remove(i)`: For any frame r, interpretation and reification of the pre-condition results in sets of heaps with the form $\{\mathrm{L} \mapsto [pl_1 \otimes i \otimes pl_2]\} \sqcup h$ where h contains enough resource to evaluate e. Notice that the value of variable `i` is always present in the list, which has been filled in by the completion process or the frame r with arbitrary partial lists. The atomic action of `remove` updates the list removing i, expressly maintaining pl_1 and pl_2. Interpretation and reification of the post-condition gives heaps of the form $\{\mathtt{i} \mapsto i, \mathrm{L} \mapsto [pl_1 \otimes pl_2]\} \sqcup h$, as required.
3. `i := getFirst()`: There are two cases for `getFirst`. For any frame r, interpretation and reification of $\mathtt{i} \rightarrow \text{-} * \mathrm{L} \mapsto [\mathrm{I} \otimes \alpha]$ results in sets of heaps with the form $\{\mathtt{i} \mapsto j, \mathrm{L} \mapsto [i \otimes pl]\} \sqcup h$. The atomic action of the `getFirst` command on these heaps is to assign the value of i to variable `i`; no other update is performed. Interpretation and reification of the post-condition using the same frame r results in heaps of the form $\{\mathtt{i} \mapsto i, \mathrm{L} \mapsto [i \otimes pl]\} \sqcup h$, exactly the outcome of the action.

   The second pre-condition, $\mathtt{i} \rightarrow \text{-} * \mathrm{L} \mapsto [\varnothing]$, results in heaps of the form $\{\mathtt{i} \mapsto j, \mathrm{L} \mapsto [\varnothing]\} \sqcup h$. To avoid an angelic axiom, soundness requires that these forms be different from that of the first pre-condition, which is evidently true (one contains only lists with at least one element, the other only empty lists). The argument then proceeds as in the first case.

□

3.3.5. Assertions and axioms for trees

As we did when defining the underlying data algebra, we follow the pattern used for our list example when defining assertions and axioms for other structures. We first defining logical variables and expressions, then the data assertions and interpretations, finally giving the command axioms. This pattern will be adopted in all future examples.

The logical expression |E| allows the calculation of the length of a tree sibling list. This is required to axiomatise the `getChild` command.

Definition 60 (Logical values and expressions for trees)**.** Given a set of addresses ADDRS (definition 43) and abstract trees ABSTREES (definition 39) , the set of **logical values for trees** LVALS is:

$$\text{LVALS} = \mathbb{Z} \cup \mathbb{B} \cup \text{ADDRS} \cup \text{TREES}$$

The set of logical expressions and their interpretations are those provided by framework (parameter 14), along with a *sibling list length* expression: for all $\text{E} \in \text{LEXPRS}$

$$|\text{E}|$$

The interpretation of this expression is:

$$(\!|\,|\text{E}|\,|\!)(\Gamma) \triangleq \begin{cases} len(t) & \text{if } (\!|\text{E}|\!)(\Gamma) = t, t \in \text{TREES} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 61 (Assertions on trees)**.** Given a set of logical expressions LEXPRS (definition 14), the set of tree specific data assertions are defined as: for all $\text{E} \in \text{LEXPRS}$

$$\begin{array}{rll} \psi ::= & \text{E}[\psi] & \text{Node} \\ | & \varnothing & \text{Empty tree} \\ | & \psi_1 \otimes \psi_2 & \text{Tree concatenation} \\ | & \alpha & \text{Body address} \end{array}$$

The assertion interpretations are:

$$\begin{array}{rl} (\!|\text{E}[\psi]|\!)^\Gamma & \triangleq \{(\!|\text{E}|\!)(\Gamma)[\mathbf{t}] \mid \mathbf{t} \in (\!|\psi|\!)^\Gamma\} \\ (\!|\varnothing|\!)^\Gamma & \triangleq \{\varnothing\} \\ (\!|\psi_1 \otimes \psi_2|\!)^\Gamma & \triangleq \{\mathbf{t}_1 \otimes \mathbf{t}_2 \mid \mathbf{t}_1 \in \psi_1, \mathbf{t}_2 \in \psi_2\} \\ (\!|\alpha|\!)^\Gamma & \triangleq \{\mathbf{x} \mid \mathbf{x} = \Gamma(\alpha)\} \end{array}$$

where, if the result would not be an abstract tree (definition 39), it is undefined.

With these assertions, we can axiomatise the tree commands.

Definition 62 (Tree update command axioms)**.** The syntactic axioms for the commands of the tree module given in definition 34 are as follows: for all $P \in \textsc{Assts}$, P exact

$$\{\texttt{nID} \to \text{-} * \text{R} \mapsto \alpha\}$$
$$\texttt{nID := createNode()}$$
$$\{\exists \text{N}.\ (\texttt{nID} \to \text{N} * \text{R} \mapsto \alpha \otimes \text{N}[\varnothing])\}$$

$$\{\texttt{pID} \to \text{P} * \texttt{cID} \to \text{C} * \alpha \mapsto \text{P}[\beta] * \gamma \mapsto \text{C}[\text{T} \wedge \mathit{is\_complete}]\}$$
$$\texttt{appendNode(pID, cID)}$$
$$\{\texttt{pID} \to \text{P} * \texttt{cID} \to \text{C} * \alpha \mapsto \text{P}[\beta \otimes \text{C}[\text{T}]] * \gamma \mapsto \varnothing\}$$

$$\{(\texttt{pID} \to \text{P} * \texttt{nID} \to \text{-} * \alpha \mapsto \text{P}[\text{T} \otimes \text{N}[\beta] \otimes \gamma] * P) \wedge e \Rightarrow \text{I} \wedge |\text{T}| = \text{I}\}$$
$$\texttt{nID := getChild(pID, } e\texttt{)}$$
$$\{\texttt{pID} \to \text{P} * \texttt{nID} \to \text{N} * \alpha \mapsto \text{P}[\text{T} \otimes \text{N}[\beta] \otimes \gamma] * P\}$$

$$\{(\texttt{pID} \to \text{P} * \texttt{nID} \to \text{-} * \alpha \mapsto \text{P}[\text{T}] * P) \wedge e \Rightarrow \text{I} \wedge \text{I} < 0 \vee |\text{T}| \leq \text{I}\}$$
$$\texttt{nID := getChild(pID, } e\texttt{)}$$
$$\{\texttt{pID} \to \text{P} * \texttt{i} \to \text{I} * \texttt{nID} \to 0 * \alpha \mapsto \text{P}[\text{T}] * P\}$$

$$\{\texttt{nID} \to \text{N} * \alpha \mapsto \text{N}[\text{T} \wedge \mathit{is\_complete}]\}$$
$$\texttt{removeSubtree(nID)}$$
$$\{\texttt{nID} \to \text{N} * \alpha \mapsto \varnothing\}$$

The semantic axioms are these syntactic axioms interpreted through all logical environments.

The axiom for `createNode` uses an existentially quantified node identifier for the newly created node. This ensures uniqueness in the same way it did for allocation in primitive heaps (example 16). We do not know *what* additional nodes will be present in the tree, as they have been "set aside" leaving only body address α. However, as all trees are finite, there will infinitely many unused node identifiers, one of which will be bound to N. The `appendNode` pre-condition uses the *is_complete* predicate for the same reason as the pre-condition of list `append` - we must be sure that no sub-data has been abstractly allocated. However, here, the reason is that we must ensure that the parent node is not a descendent of the child. By ensuring that the sub-tree of child C is complete, and by knowing that the parent P is allocated in abstract cell α, the separating conjunction assures us that relationship is impossible.

Here, the analysis of data captured in logical variables is used to determine which case

of `getChild` is appropriate. We must be assured that node N is the I^{th} child. This is accomplished by using the logical expression for sibling length calculation on the prefix siblings-list captured in T.

Lemma 11 (Atomic soundness of tree commands)**.** *Given the views* VIEWS *from a structural addressing algebra on abstract trees (definition 42 and definition 39 respectively), the tree commands are atomically sound with respect to their axioms (definition 62), thus satisfying the atomic soundness requirement of the framework (definition 12).*

We elide the proof, as it is similar in structure to previous atomic soundness proofs (e.g. example 16). The two interesting facets are in `appendNode` and `removeChild`. The append command uses pre-condition $\{\texttt{i} \rightarrow \text{I} * \alpha \mapsto \text{P}[\beta] * \gamma \mapsto \text{C}[\text{T} \wedge \mathit{is\_complete}]\}$. Notice that the children of node C must be complete (that is, have no abstractly allocated sub-tree). This is required to show that the parent node is not a sub-tree of the child (which would create a cycle). We will examine this case in more detail in chapter 7.

In the pre-condition for `removeSubtree`, we also capture the entire sub-tree and ensure that no body addresses are present. This is so that destroying the tree does not destroy a body address. Were it possible to destroy a body address, the axiom could invalidate certain frames. Consider the alternative pre-condition $\texttt{nID} \rightarrow \text{N} * \alpha \mapsto \text{N}[\beta]$. By picking the frame $\beta \mapsto \text{M}[\varnothing]$, this defective axiom would transform $\texttt{nID} \rightarrow \text{N} * \alpha \mapsto \text{N}[\beta] * \beta \mapsto \text{M}[\varnothing]$ into $\texttt{nID} \rightarrow \text{N} * \alpha \mapsto \varnothing * \beta \mapsto \text{M}[\varnothing]$. The completions of the pre-condition always have a node labelled M as the child of node N, so there can never be a node M elsewhere in the tree. Completions of the post-condition under this frame would *have to* to generate a location for β, which would ensure that every tree contained a node M. But, the action of the command is to destroy N and all of its children, so the post-condition should *never* contain a node M! By destroying the β body address, we would have invalidated any frames containing a β cell, which is counter to atomic soundness.

3.3.6. Proofs of examples

We now have all the machinery to prove programs using our abstract list and tree libraries. We show the admissible commands given in sections 3.1.1 and 3.1.2 have their natural specifications. For lists, the commands of figure 3.8 have the following specifications:

$$\{\texttt{i} \to \text{-} * \text{L} \mapsto [\text{A} \wedge \mathit{is\_complete}]\}$$
$$\texttt{i := getLast()}$$
$$\{\exists \text{I}.\ (\texttt{i} \to \text{I} * \text{L} \mapsto [\text{A} \wedge \mathit{is\_complete}] \wedge ((\exists \text{B}.\ \text{A} = \text{B} \otimes \text{I}) \vee (\text{A} = \varnothing \wedge \text{I} = 0)))\}$$

$$\{\texttt{k} \to \text{K} * \texttt{b} \to \text{-} * \text{L} \mapsto [\text{A}]\}$$
$$\texttt{b := contains(k)}$$
$$\{\texttt{k} \to \text{K} * ((\text{K} \in \text{A} \wedge \texttt{b} \to \text{TRUE}) \vee (\text{K} \notin \text{A} \wedge \texttt{b} \to \text{FALSE})) * \text{L} \mapsto [\text{A}]\}$$

The `getLast` specification is perhaps surprising. A suitable small axiom would include the list resource $\text{L} \mapsto [\alpha \otimes \text{I}]$, that is, the last element only. However, we are using an *implementation* here, which requires all the previous elements in order to find the last. This shows that, even when implementable, the specifications for some commands may be smaller when given by axiomatisation.

In giving the programs, we introduced a local variable construction to ease the examples. The corresponding reasoning rule is below. It is applicable only if $P * \texttt{p1} \to \text{-} * \cdots * \texttt{pn} \to \text{-}$ is satisfiable (that is, if P does not already mention the newly local variables).

$$\frac{\{P * \texttt{p1} \to \text{-} * \cdots * \texttt{pn} \to \text{-}\}\ \mathbb{C}\ \{Q * \texttt{p1} \to \text{-} * \cdots * \texttt{pn} \to \text{-}\}}{\{P\}\ \texttt{local p1, ..., pn}\ \{\mathbb{C}\}\ \{Q\}}$$

The proof that `getLast` satisfies this specification is in figure 3.17; the proof for `contains` is in the appendices, section A.1.1. We highlight a small part of the proof in detail, as it outlines an idiom common to many structural separation logic proofs. The first step is *case analysis*, where logical variables capturing structure are broken down into their possibilities. In this case, A may or may not be an empty list, so we consider both possibilities with the rule of disjunction. If it is empty, we apply the "empty list" instance of the `getFirst` axiom. Otherwise, we must use abstract allocation to localise the surplus data. It can then be framed off, and the axiom applied.

This pattern is very common in all our program proofs. We first manipulate the data representation to *localise* the data of interest to a command. This often involves disjunctions of possibilties. We then apply abstract allocation to set aside unneeded data. Finally, we apply an axiom, then reverse the previous steps with abstract deallocation. In general, we typically elide many of these steps as straightforward, much as separation logic proof sketches do not detail each use of entailment in unrolling inductive predicates.

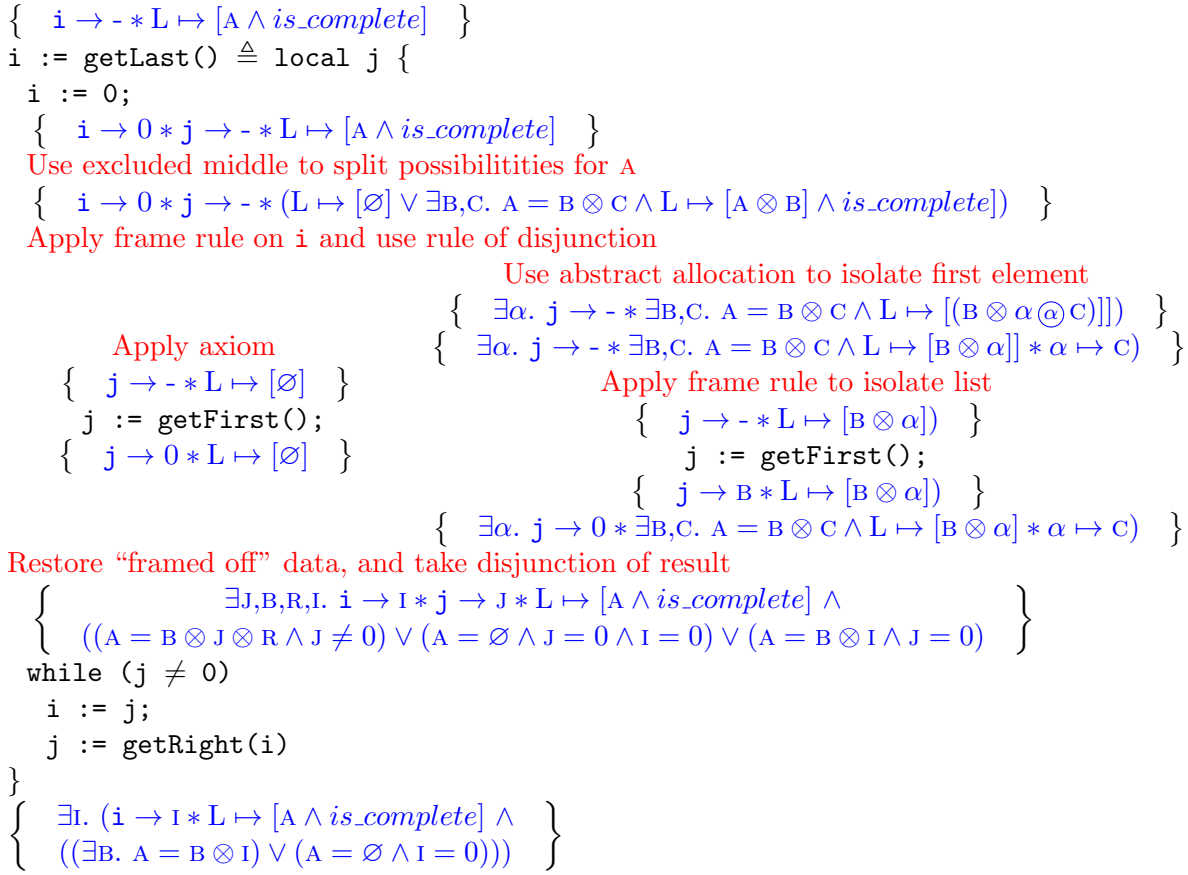


Figure 3.17.: Proof that `getLast` satisfies its specification

The tree command of figure 3.8 has the following specification:

$$\{\texttt{n} \to \text{N} * \texttt{c} \to \text{-} * \alpha \mapsto \text{N}[\text{T}] \wedge |\text{T}| = \text{C}\}$$
$$\texttt{c := count(n)}$$
$$\{\texttt{n} \to \text{N} * \texttt{c} \to \text{C} * \alpha \mapsto \text{N}[\text{T}] \wedge |\text{T}| = \text{C}\}$$

The proof of this specification, which uses the idiom detailed above, is below. The complexity of the loop invariant results from the two possibilities at each loop interaction. Broken down, it contains:

- $\texttt{n} \to \text{N} * \texttt{c} \to \text{E} * \texttt{d} \to \text{D}$ contains the state of the program variables, bound to logical variables.
- $\alpha \mapsto \text{N}[\text{T}] \wedge |\text{T}| = \text{C}$: Describes the tree resource we currently have: the node N whose children are being counted, and a logical variable T that captures this resource. The length of this tree is captured in logical variable C. Because the length function $|\text{T}|$ has returned a value, we can infer that the list must contain *no* body addresses (as the function $|\cdot|$ is undefined in those cases.
- The two possibilities for relating the length C of the data captured in T to the variables. Either:
  - $\text{D} = 0 \wedge \text{E} = \text{C}$ The identifier returned by the last call to `getChild` is 0, thus the current counter variable must contain the length of the list; or
  - $\text{D} \neq 0 \wedge \exists \text{B,R}.\ \text{T} = \text{B} \otimes \text{D} \otimes \text{R} \wedge \text{E} = |\text{B}|)$ The identifier returned by the last call to `getChild` was not 0. Thus, we can break the list down into a prefix B, followed by the element returned D, followed by the rest of the list R. The length of the prefix must have been assigned to the counter variable.

As in the `getLast` proof, the full text would require case analysis of each possibility. We elide these cases, as they are straightforward to perform, providing little insight into the reasoning.

$$\left\{ \quad \mathtt{n} \to \mathrm{N} * \mathtt{c} \to \text{-} * \alpha \mapsto \mathrm{N}[\mathrm{T}] \wedge |\mathrm{T}| = \mathrm{C} \quad \right\}$$

```
c := count(n) ≜ local d {
  c := 0;
  d := getChild(n, c)
```

$$\left\{ \begin{array}{l} \exists \mathrm{D}.\ \mathtt{n} \to \mathrm{N} * \mathtt{c} \to 0 * \mathtt{d} \to \mathrm{D} * \alpha \mapsto \mathrm{N}[\mathrm{T}] \wedge \\ |\mathrm{T}| = \mathrm{C} \wedge ((\mathrm{C} = 0 \wedge \mathrm{D} = 0) \vee (\mathrm{C} > 0 \wedge \mathrm{D} \neq 0)) \end{array} \right\}$$

$$\left\{ \begin{array}{l} \exists \mathrm{D},\mathrm{E}.\ \mathtt{n} \to \mathrm{N} * \mathtt{c} \to \mathrm{E} * \mathtt{d} \to \mathrm{D} * \alpha \mapsto \mathrm{N}[\mathrm{T}] \wedge \\ |\mathrm{T}| = \mathrm{C} \wedge ((\mathrm{D} = 0 \wedge \mathrm{E} = \mathrm{C}) \vee (\mathrm{D} \neq 0 \wedge \exists \mathrm{B},\mathrm{R}.\ \mathrm{T} = \mathrm{B} \otimes \mathrm{D} \otimes \mathrm{R} \wedge \mathrm{E} = |\mathrm{B}|)) \end{array} \right\}$$

```
  while (d ≠ 0)
    c := c + 1;
    d := getChild(n, c)
}
```

$$\left\{ \quad \mathtt{n} \to \mathrm{N} * \mathtt{c} \to \mathrm{C} * \alpha \mapsto \mathrm{N}[\mathrm{T}] \wedge |\mathrm{T}| = \mathrm{C} \quad \right\}$$

Weakest pre-conditions

As discussed in section 2.3.4, we can give weakest pre-conditions for our frame using the *magic flower*. This is necessary in structural separation logic, as normal entailment is cannot re-express data using abstract addresses. For example, using the natural lifting of $\preccurlyeq\!\!*$ to the syntactic assertion language, the weakest pre-conditions for `createNode()` from our tree example is:

$$\{\exists \alpha, \mathrm{N}.\ ((\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \mathtt{nID} \to \mathrm{N}) \preccurlyeq\!\!* Q) * \mathrm{R} \mapsto \alpha * \mathtt{nID} \to \mathrm{J}\}$$
$$\mathtt{nID := createNode()}$$
$$\{Q\}$$

The derivation, starting with the axiom, is:

$$\dfrac{\dfrac{\dfrac{\{\mathrm{R} \mapsto \alpha * \texttt{nID} \to \mathrm{J}\}}{\texttt{nID := createNode()}}}{\{\exists \mathrm{N}.\ (\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N})\}}}{\{((\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N}) \preccurlyeq\!\!* Q) * \mathrm{R} \mapsto \alpha * \texttt{nID} \to \mathrm{J}\}}$$
$$\texttt{nID := createNode()}$$
$$\dfrac{\{((\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N}) \preccurlyeq\!\!* Q) * \exists \mathrm{N}.\ (\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N})\}}{\{\exists \alpha, \mathrm{N}.\ ((\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N}) \preccurlyeq\!\!* Q) * \mathrm{R} \mapsto \alpha * \texttt{nID} \to \mathrm{J}\}}$$
$$\texttt{nID := createNode()}$$
$$\dfrac{\{\exists \alpha.\mathrm{N}.\ ((\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N}) \preccurlyeq\!\!* Q) * (\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N})\}}{\{\exists \alpha, \mathrm{N}.\ ((\mathrm{R} \mapsto \alpha \otimes \mathrm{N}[\varnothing] * \texttt{nID} \to \mathrm{N}) \preccurlyeq\!\!* Q) * \mathrm{R} \mapsto \alpha * \texttt{nID} \to \mathrm{J}\}}$$
$$\texttt{nID := createNode()}$$
$$\{Q\}$$

Notice that, were $-\!\!*$ used rather than $\preccurlyeq\!\!*$, the application of the consequence rule could not allocate and deallocate the α addressed sub-data. The only post-conditions Q that would work would be those with α already extant. It is clear that `createNode` can always be safely used, so that pre-condition would *not* be the weakest in any useful sense.

Similar weakest pre-conditions can be constructed for the list library, and other tree commands.

3.3.7. Rule of conjunction

Recall the requirement of conjunction, primitive conjunctivity, given in definition 16, which were dependent on the choice of primitive reification function. The primitive reification function for abstract heaps (definition 47) does not satisfy these requirements. Therefore, when using semantic consequence or semantic strengthening to perform abstract allocation, the rule of conjunction is not sound.

Lemma 12 (The conjunction rule is unsound)**.** *The rule of conjunction is unsound when reasoning about abstract heaps.*

Proof. By counter-example. Consider the tree assertion $\alpha \mapsto a[b[\varnothing]]$ and the program `skip`. This program run with that assertion as a pre-condition evidently terminates. Assume that the rule of conjunction is sound. Then, the following two derivations are both valid via the rules of semantic consequence and skip:

$$\begin{array}{ll}
\left\{\quad \alpha \mapsto a[b[\varnothing]] \quad\right\} & \left\{\begin{array}{l} \alpha \mapsto a[b[\varnothing]] \\ \exists\beta.\ (\alpha \mapsto a[\beta] * \beta \mapsto b[\varnothing]) \end{array}\right\} \\
\texttt{skip} & \texttt{skip} \\
\left\{\quad \alpha \mapsto a[b[\varnothing]] \quad\right\} & \left\{\begin{array}{l} \exists\beta.\ (\alpha \mapsto a[\beta] * \beta \mapsto b[\varnothing]) \\ \exists\beta.\ (\alpha \mapsto a[\beta] * \beta \mapsto b[\varnothing]) \end{array}\right\}
\end{array}$$

By the rule of conjunction, the following triple must hold:

$$\{\alpha \mapsto a[b[\varnothing]] \wedge \alpha \mapsto a[b[\varnothing]]\}\ \texttt{skip}\ \{\alpha \mapsto a[b[\varnothing]] \wedge \exists\beta.\ (\alpha \mapsto a[\beta] * \beta \mapsto b[\varnothing])\}$$

However, whilst the pre-condition of this is defined (and equal to $\alpha \mapsto a[b[\varnothing]]$), the post-condition is unsatisfiable. Even though both haves have the same reification, they are *different* instrumented heaps. Therefore, there are no heaps in common between the first conjunct (describing all heaps with one address pointing to $a[b[\varnothing]]$) and the second (which splits the data across two cells). This means the program must not terminate. But, `skip` always terminates. □

The unsoundness of the conjunction rule here is not surprising. Abstract allocation is inherently *angelic*; the choice of where and when to split is given to the constructor of the proof. There are many different ways to cut up the same data, and so the uses of abstract allocation and deallocation may differ between two proofs of the same program, and so the post-conditions may describe abstract heaps with different structural addresses.

3.4. Comparison to previous structured data reasoning

In section 1.2, we gave an overview of previous work that enables local reasoning about structured data. Here, we more directly compare these methods with structural separation logic, indicating the relative strengths and weaknesses. We will focus on the `appendNode` tree library command given in section 3.1.2. This one command demonstrates many of the challenges of structured data: abstract data representation of trees, localisation of sub-data in terms of the nodes being considered, and consideration of relationships between those nodes (demonstrating that the parent is not an ancestor of the child).

We will compare the proof of the single command `appendNode(p, c)` using structural separation logic, inductive predicates, abstract predicates, and context logic. In all three cases, the proof consists of three steps:

1. **Data localisation**: The representation of the tree is manipulated so that the sub-tree `p` and `c` are *separate* from other data. The irrelevant tree structure must be "framed off", ready for the application of the `appendNode` axiom.
2. **Axiom application**: The axiom is applied, and the representation updated.
3. **Data unification**: The "framed off" data is restored, and the representation of the tree is manipulated to reconnect the sub-data with the surrounding data.

As seen in definition , structural separation logic uses an entirely abstract data representation. The localisation phase consists of identifying the sub-data using composition operator, $d_1 \circledcirc d_2$, then applying abstract allocation and the frame rule. The axiom is a straight forward update of the data representation, using *completeness* in the data to ensure the non-ancestral relationship between the child and parent nodes hold. The derivation of the progam, using a simple example tree, is: let $V = (\mathtt{p} \to \mathrm{B} \wedge \mathtt{c} \to \mathrm{D} * E)$

$$\dfrac{\dfrac{\{V * \beta \mapsto b[\delta] * \delta \mapsto d[complete]\}\ \texttt{appendNode(p, c)}\ \{V * \beta \mapsto b[\delta \otimes d] * \delta \mapsto \varnothing\}}{\{V * \exists \beta, \gamma, \delta.\ \alpha \mapsto a[\beta \otimes \delta] * \beta \mapsto b[\delta] * \gamma \mapsto c * \delta \mapsto d[complete]\}\ \texttt{appendNode(p, c)}\ \{V * \exists \beta, \gamma, \delta.\ \alpha \mapsto a[\beta \otimes \delta] * \beta \mapsto b[\delta \otimes d] * \gamma \mapsto c * \delta \mapsto \varnothing\}}}{\{V * \alpha \mapsto a[b[c] \otimes d]\}\ \texttt{appendNode(p, c)}\ \{V * \alpha \mapsto a[b[c \otimes d]]\}}$$

3.4.1. Inductive predicates

We now compare the above to inductive predicates. The standard technique ([63], [69]) for representing a structured data using inductive predicates is to link an *algebraic* tree repesentation (such as the abstract tree of definition 39) to an underlying heap structure. By picking a heap representation, we can give a *tree* predicate that represents the algebraic structure via a set of heap cells with varying pointer relationships. The difficulty comes in *localising* the data into smaller sub-tree representations. This is accomplished by unfolding the predicate, revealing implementation details. These details, typically an underlying heap structure, are either manipulated directly or are repackaged into predicates representing the sub-data.

One typical example of a tree representation is the *five-pointer* style. Each tree node is represented with a heap cell of five fields: left, right, up, first child and last child. The address of the heap cell is punned as the identifier of the node. The left and right pointers reference the nodes directly left and right siblings (and are null if the node is the first or last of a sibling list). The up field references the nodes parent (and is null if the node is a

root), and the first and last child fields reference the first and last child in a linked list of children. The $tree(\mathtt{x}, \mathrm{T})$ predicate then states that algebraic tree T is represented in the heap starting at address `x` can be given as: let $first$ and $last$ extract the identifier of the first and last element of a node list

$$
\begin{array}{ll}
tree(\mathtt{x}, \mathrm{T}) & \triangleq node(t, 0, 0, 0) \\
node(\mathrm{N}[\mathrm{T}], l, r, u) & \triangleq \begin{array}{l} first(\mathrm{T}) = \mathrm{FC} \wedge last(\mathrm{T}) = \mathrm{LC} \wedge \\ \mathrm{N} \mapsto \mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{FC}, \mathrm{LC} * nodelist(\mathrm{T}, \mathrm{N}, 0, 0) \end{array} \\
nodelist(\mathrm{N}[\mathrm{T}], l, r, u) & \triangleq node(\mathrm{N}[\mathrm{T}], l, r, u) \\
nodelist(\mathrm{T} \otimes \mathrm{U}, l, r, u) & \triangleq \exists \mathrm{I}.\ nodelist(\mathrm{T}, l, \mathrm{I}, u) * nodelist(\mathrm{U}, \mathrm{I}, r, u) \\
nodelist(\varnothing, l, r, u) & \triangleq emp \wedge l = 0 \wedge r = 0
\end{array}
$$

Whilst standard, this representation is unwieldy. Notice that a *node* predicate represents the resource for a node and *all* its children. To represent a node in the tree *without* capturing the entire sub-tree, we must include an additional "solo" node predicate. Such a predicate will be required to represent the parent node in `appendNode(p, c)`, as the child node `c` may be a descendent of P, and so must not be captured in the representation of P.

$$solo\_node(n, l, r, u, fc, lc) \quad \triangleq \mathrm{N} \mapsto \mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{FC}, \mathrm{LC}$$

We these, we might assume we can axiomatise the `appendNode(p, c)` command as by including the solo node for the parent `c`, the entire tree for `c`, and solo node for the last child of `p` (which must be updated to have `c` hooked to the end of it, giving a pre-condition like:

$$V * solo\_node(\mathrm{P}, \mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{FC}, \mathrm{LC}) * node(\mathrm{C}, \mathrm{CL}, \mathrm{CR}, \mathrm{CU}) * solo\_node(\mathrm{LC}, \mathrm{LCL}, \mathrm{LCR}, \mathrm{LCU}, \mathrm{LCFC}, \mathrm{LCLC})$$

Alas, even this is not sufficient: we must include the parent of the `c` node, to update its child pointers, and we must consider the case where `c` is already a child of `p`. We do not consider these here as these problems, whilst tedious, are surmountable. The critical point is that even once an inductive predicate for the tree is devised, we have only a data representation. We can give an axiom to `appendNode`, but to use this axiom, we must still localise the data of interest inside a representation. Thus, we must repeatedly unfold, and refold the predicates, until we have represented `p` via the *solo_tree* predicate, `c` using the *node* predicate. One typical step will be extracting a *solo_node* predicate from a *node* predicate by unfolding the *node* one step, and folding the heap cell exposed into *node*:

$$node(\text{N}[\text{T}], l, r, u) \iff solo\_node(\text{N}, l, r, u, first(\text{T}), last(\text{T})) * nodelist(\text{T}, \text{N}, 0, 0)$$

This rewriting is both tedious and error prone. Therefore, comparing structural separation logic to inductive predicates we see that inductive predicates:

1. Allow verification of implementation: As the representation can be broken open to reveal the underlying heap structure, inductive predicates allow the underlying implementation code to be verified with respect to the representation. Structural separation logic, operating entirely on abstract representations, does not immediately allow this. However, with Raad [59], we are currently working on implementation verification via refinement.
2. Suffer from difficult localisation: The folding and unfolding of the predicates is tedious and error prone when seen against the comparatively simple approach of abstract allocation.
3. Break abstraction: Data localisation requires unfolding the *tree* predicate in order to extract the sub-data. This ties client proofs to the specific implementation of the tree structure. Structural separation logic, working entirely on abstract data, does not suffer from this problem.
4. Use non-uniform presentation: The correct choice of unfolding for the predicates to localise the sub-data is innately tied to the choice of predicates. Thus, the techniques of data localisation differ from library to library. Structural separation logic offers abstract allocation as a uniform approach to localising data.

3.4.2. Abstract predicates

Abstract predicates restore abstraction to the presentation of inductive predicates by hiding the predicate bodies from client programs. However, this means predicates cannot be folded and unfolded directly. Data localisation must thus be provided by *axioms*, which then must be justified by any underlying implementation.

We can consider abstract predicates simply by erasing the right-hand sides from the inductive predicates in section 3.4.1. One localisation axiom is the unfolding of the *node* predicate into *solo_node* and *nodelist*, but one must include many more, as we have to provide such an axiom for every splitting a proof may want. The *signatures* of these predicates, in terms of the parameters they take, must be sufficient to allow these axioms to unambiguously split and rejoin data. Thus, the parameters on, for example,

$node(\text{N}[\text{T}], l, r, u)$, leak some of the pointer implementation details into the abstraction. Ever more clever choices of predicates can be devised to minimise this, but due to the underlying need to be justified by a heap structure, this problem always remains. Thus, in comparison to structural separation logic, abstract predicates:

1. Allows verification of several implementations: Abstract predicate do hide some implementation concerns from clients, allowing several implementations to be verified. We hope our upcoming refinement approach will allow a similar result for structural separation logic.
2. Non-uniform presentation: As with inductive predicates, the choice of representations is ad-hoc, and the localisation created on a structure by structure basis.
3. Implementation details leak into predicate signatures: The parameters on "abstract" predicates are needed to ensure that predicates representing sub-data can be reconnected with the predicates representing super-data. These parameters must be sufficient for any underlying implementation to reconnect the data. Thus, the number and choice of parameters is directly linked to expected implementations, whilst offering minimal value to the abstraction.

Comment 7. This final point is perhaps the key problem solved by structural separation logic. We allow predicates to be split and reconnected using the mechanism of structural addresses. It is reasonable to see structural separation logic as the first step in a new theory of abstract predicates where *separation* and *connectivity* is abstracted, rather than just *representation*.

3.4.3. Context logic

Context logic is much closer to the level of abstraction we can achieve with structural separation logic. It avoids all implementation considerations by operating directly on the abstract representation of the structure. The key difference from structural separation logic is the data localisation phase, where separation is now context application. For example, the derivation for our example program is:

$$\frac{\{V_c \wedge C \circ_{\mathrm{X}} b[c] \circ_{\mathrm{Y}} d\}\ \texttt{n := appendNode(p, c)}\ \{V_c \wedge C \circ_{\mathrm{X}} b[c \otimes d] \circ_{\mathrm{Y}} \varnothing\}}{\begin{array}{c}\{V_c \wedge C \circ_{\mathrm{X}} b[c] \circ_{\mathrm{Y}} da[b[c] \otimes d]\} \\ \texttt{n := appendNode(p, c)} \\ \{V_c \wedge C \circ_{\mathrm{X}} b[c \otimes d] \circ_{\mathrm{Y}} \varnothing\}\end{array}}$$
$$\{V_c \wedge a[b[c] \otimes d]\}\ \texttt{n := appendNode(p, c)}\ \{V_c \wedge a[b[c \otimes d]]\}$$

Separation can only occur in an order, depth last manner. In comparison to structural separation logic, context logic:

1. **Has non-commutative separation**: The separating application connective is not commutative, meaning data must be extracted "depth-last" (with the deepest sub-data coming at the end). This means one cannot reorder the representation of sub-data to fit specific axioms or reasoning rules. Most importantly, this means one cannot give a parallel composition rule. It also results in larger axioms, viz: the covering context needed to handle `appendNode` here (explained in detail in section 1.2.3).
2. **Cannot directly co-exist with heaps**: Separating application ensures that the data presentation is not simply a bag of heap cells. Thus, normal heap style reasoning cannot naturally co-exist with the context structured data. One problematic consequence of this is having more than instance of a data structure is difficult, such as a list and a tree together. Structural separation logic, by reusing the standard heap mechanism, allows heterogeneous data to be combined into a single reasoning system.

3.4.4. Overview of advantages

Structural separation logic unifies the natural abstract reasoning of context logic with the natural "bag" model of separation. Taking the most useful parts of each pre-existing technique, it offers improvements in:

- Abstraction of implementation: The specific choices of an *implementation* are entirely hidden, with only the abstract structure exposed in proofs.
- Abstraction of separation: The connectivity information used by the underlying data structures, such as the pointers in sub-section 3.4.1, are hidden, with data separability abstracted via a uniform equational theory of *compression*.
- Natural folding and unfolding behaviour: The localising of sub-data is achieved via a simple *abstract allocation*.

- Separation-logic style composition: The composition operation of $*$ both commutes and associates. This enables abstract heaps to co-exist with the standard heaps of other separation logic techniques.

3.5. Extensions and alternatives

We now give a short summary of alternative choices and extensions that can be made in the development of structural separation logic.

3.5.1. The generalised frame rule

We now examine how the *generalised frame rule* of the views framework allows an alternate method of performing abstract allocation and deallocation. This allows us to relate our approach to the reasoning of segment logic [38, 70]. We also extend views with a *relational frame rule*, allowing us to compare our approach of semantic consequence with this alternate choice.

Recall the frame rule of theorem 1. The views framework [22] provides a *generalised* version of this rule that can apply functions to the pre- and post-condition. These *generalised frame functions* functions must satisfy certain properties.

Definition 63 (Generalised frame functions)**.** Given an addressed value view $(\textsc{Views}, [\![\cdot]\!], \circledast, 0)$ (parameter 8), transition labels $\textsc{ActionLbls}$ (definition 5) and atomic command actions $[\![\langle \textsc{c} \rangle]\!]$ (parameter 6) a function $f : \textsc{Views} \to \textsc{Views}$ is a **generalised frame function** if it preserves all actions under all frames:

$$\forall p, q \in \textsc{Views}, \alpha \in \textsc{ActionLbls}. \quad \begin{array}{c} (\forall r \in \textsc{Views}.\ [\![\langle \alpha \rangle]\!] \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor) \\ \Longrightarrow \\ (\forall r \in \textsc{Views}.\ [\![\langle \alpha \rangle]\!] \lfloor f(p) * r \rfloor \subseteq \lfloor f(q) * r \rfloor) \end{array}$$

Given a generalised frame function, we can generalise the frame rule.

Proposition 2 (Generalised frame rule)**.** Given an addressed value view $(\textsc{Views}, [\![\cdot]\!], \circledast, 0)$ (parameter 8) and the semantic triple judgement (definition 14) if, given $p, q \in \textsc{Views}$, $\vDash \{p\}\ \mathbb{C}\ \{q\}$ holds then for all generalised frame functions f, the semantic triple judgement $\vDash \{f(p)\}\ \mathbb{C}\ \{f(q)\}$ holds.

By defining a family of frame functions $f_r(p) = p * r$ for all r, it is evident the generalised frame rule subsumes the standard frame rule. It also admits function with very "non-frame" like behaviour. We can define a generalised frame function for abstract heaps that performs a single-step collapse on a structural address.

Definition 64 (Hiding frame function)**.** The **hiding frame function** for address $\mathbf{x}$, $\downarrow_{\mathbf{x}}$: Views $\rightharpoonup$ Views is defined as:

$$\downarrow_{\mathbf{x}}(p) = \begin{cases} \emptyset \text{ if } \exists \mathbf{h} \in p. \quad \nexists \mathbf{h}'.\ \mathbf{h} \downarrow \mathbf{h}', \mathrm{dom}(\mathbf{h}') = \mathrm{dom}(\mathbf{h}) \setminus \{\mathbf{x}\} \\ \{\mathbf{h}' \mid \mathbf{h} \in p, \mathbf{h} \downarrow \mathbf{h}', \mathrm{dom}(\mathbf{h}') = \mathrm{dom}(\mathbf{h}) \setminus \{\mathbf{x}\}\} \text{ otherwise} \end{cases}$$

Lemma 13 (The hiding frame function is a frame function)**.** *The hiding frame function of definition 64 is a generalised frame function in the sense of definition 63.*

Proof. Note that the hiding frame function performs exactly one step of the collapse process, or (if there is no abstract address to use), produces the empty view. As reification is defined in terms of collapse, the result then follows by the definition of reification and theorem 3, which ensures that $\lfloor f(p) * r \rfloor = \lfloor p * r \rfloor$ and $\lfloor f(q) * r \rfloor = \lfloor q * r \rfloor$, and the fact that the empty view reifies to the empty set (ensuring the implication holds vacuously if an invalid address is chosen for hiding). □

The hiding frame function obviates the need to use semantic consequence for abstract allocation and deallocation. Assuming the straightforward lift of $\downarrow_{\mathbf{x}}$ to a syntactic assertion, it enables a proof structures like:

$\{\alpha \mapsto 2\}$	`remove(2)`	$\{\alpha \mapsto \varnothing\}$	*Axiom*
$\{\mathrm{L} \mapsto [3 \otimes \alpha \otimes 5] * \alpha \mapsto 2\}$	`remove(2)`	$\{\mathrm{L} \mapsto [3 \otimes \alpha \otimes 5] * \alpha \mapsto \varnothing\}$	*Gen. frame*
$\{\downarrow_\alpha (\mathrm{L} \mapsto [3 \otimes \alpha \otimes 5] * \alpha \mapsto 2)\}$	`remove(2)`	$\{\downarrow_\alpha (\mathrm{L} \mapsto [3 \otimes \alpha \otimes 5] * \alpha \mapsto \varnothing)\}$	*Consequence*
$\{\mathrm{L} \mapsto [3 \otimes 2 \otimes 5]\}$	`remove(2)`	$\{\mathrm{L} \mapsto [3 \otimes 5]\}$	

This proof recalls those using Wheelhouse's segment logic [38]. In fact, when lifted to a syntactic assertion of f, it gives a frame rule exactly matching the *hiding rule* of [38] (see figure 3.18).

However, proofs in this style are more rigid and carry more baggage than those using semantic consequence. They can allocate and deallocate at most one address at a time and they cannot leave data in an allocated state, as the same function must be applied to both the pre- and post-condition. Moreover, the hiding function $\downarrow_{\mathbf{x}}$ is left in the data, and

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\mathrm{H}\alpha.\ P\}\ \mathbb{C}\ \{\mathrm{H}\alpha.\ Q\}} \qquad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\downarrow_\alpha P\}\ \mathbb{C}\ \{\downarrow_\alpha Q\}}$$

Figure 3.18.: On the left, the *hiding rule* of segment logic. On the right, the collapsing frame function of definition 64 used as a generalised frame function.

must be removed by consequence. Given that, one may as well use semantic consequence to perform the entire process.

The relationship between semantic consequence and the generalised frame rule

Even though we choose semantic consequence to enable abstract allocation, proofs using the hiding frame function are sufficiently similar to warrant examining the links more closely. Most obviously, semantic consequence is a relation on pre- and post-conditions, whereas the generalised frame rule uses functions over them. This artificially restricts use of the rule. For example, it is difficult to design functions that mutate the pre-condition, but leave the post-condition untouched. To ease these restrictions, we can move to relational transformations.

We define *relational view transformers*, $\approx$, as the relational analogue of generalised frame functions. To enable a sound rule using these relations, we require some conditions. The first requirement, left totality, eliminates cases $\{p'\}\ \mathbb{C}\ \{q'\}$ where $p' \approx p$ but there is no q such that $q' \approx q$. Were such relations allowed, we could pick a relation showing the divergence of any code. The second is the natural extension of the generalised frame rule requirement.

Definition 65 (Relational view transformers)**.** Given a set of views Views, a relation $\approx \subseteq \textsc{Views} \times \textsc{Views}$ is a **relational view transformer** if it is both left total and preserves all actions under frames. That is, for all $p \in \textsc{Views}$, there exists some $q \in \textsc{Views}$ such that $p \approx q$; and

$$\forall\ p, q \in \textsc{Views},\ \alpha \in \textsc{ActionLbls}.\quad (\forall r \in \textsc{Views}.\ [\![\alpha]\!]\lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor) \implies (\forall r, p', q' \in \textsc{Views}.\ p \approx p' \wedge q \approx q' \implies [\![\alpha]\!]\lfloor p' * r \rfloor \subseteq \lfloor q' * r \rfloor)$$

With relational view transformers, we can define a relation frame rule.

Proposition 3 (Relational frame inference)**.** Assume $\approx$ is a relational view transformer, and $p, p', q, q' \in \textsc{Views}$. If $p' \approx p$, $q' \approx q$ and $\{p'\}\ \mathbb{C}\ \{q'\}$, then $\{p\}\ \mathbb{C}\ \{q\}$.

This rule is sound using reasoning similar to the proof of the normal frame rule (theorem 1), and using the properties of relational frame transformers. Note that if $\approx$ is functional, then this rule collapses to the generalised frame rule. This definition is similar to one proposed by Parkinson during development of the Views framework.

With semantic consequence, if we know $p \preccurlyeq p', q' \preccurlyeq q$, then we can justify the triple $\{p\}\ \mathrm{C}\ \{q\}$ using the triple $\{p'\}\ \mathrm{C}\ \{q'\}$. With relational view transformers, if we know $p' \approx p$ and $q' \approx q$, we can justify the same relationship between triples. The only difference between the relational frame rule and the semantic consequence rule is the order of the pre-condition relationship. Consider then relations $\mathcal{R}$ which are symmetric and reflexive.

Lemma 14 (View strengthening and weakening relations)**.** *Let $\sim \subset \textsc{Views} \times \textsc{Views}$ be a symmetric, reflexive relation. Then, $\sim$ satisfies the requirements of a semantic consequence relation if and only if it satisfies the requirements of a relational frame relation.*

Proof. **Right direction**: We first show that if $\sim$ is a semantic consequence relation, it is a relational view transformer. We thus have the following properties by assumption: ① $\sim$ is symmetric; ② $\sim$ is reflexive; and ③ $\forall p, q \in \textsc{Views}.\ p \sim q \implies \forall r \in \textsc{Views}.\ \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$.

We must show the conditions of definition 65: That $\sim$ is left total, and that it preserves actions:

$$\forall\ p, q \in \textsc{Views},\ \alpha \in \textsc{ActionLbls}.\quad (\forall r \in \textsc{Views}.\ [\![\alpha]\!]\lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor) \implies (\forall r, p', q' \in \textsc{Views}.\ p \sim p' \wedge q \sim q' \implies [\![\alpha]\!]\lfloor p' * r \rfloor \subseteq \lfloor q' * r \rfloor)$$

Left totality is straightforward, as all reflexive relations are left total. For the action preservation property, note that by symmetry, if $p \sim q$ then $q \sim p$. Hence by ③ , if $p \sim q$, then $\forall r \in \textsc{Views}.\ \lfloor p * r \rfloor = \lfloor q * r \rfloor$. Assume the premise of our goal implication, that ④ $\forall r \in \textsc{Views} [\![\alpha]\!]\lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$.

We must show $(\forall r, p', q' \in \textsc{Views}.\ p \sim p' \wedge q \sim q' \implies [\![\alpha]\!]\lfloor p' * r \rfloor \subseteq \lfloor q' * r \rfloor)$. As $p \sim p'$ and $q \sim q'$, $\lfloor p * r \rfloor = \lfloor p' * r \rfloor$, and $\lfloor q * r \rfloor = \lfloor q' * r \rfloor$ for all r. Therefore, that $[\![\alpha]\!]\lfloor p' * r \rfloor \subseteq \lfloor q' * r \rfloor$ is given by ④ .

Left direction: We now show that if $\sim$ is a relational view transformer, it is a semantic consequence relation. We thus have the following properties by assumption: ① $\sim$ is symmetric; ② $\sim$ is reflexive; ③ $\sim$ is left-total and; ④ $\forall s, t \in \textsc{Views}, \alpha \in \textsc{ActionLbls}$.

$$(\forall r \in \textsc{Views}.\ [\![\alpha]\!]\lfloor s * r \rfloor \subseteq \lfloor t * r \rfloor) \implies$$
$$(\forall r, s', t' \in \textsc{Views}.\ s \sim s' \wedge t \sim t' \implies [\![\alpha]\!]\lfloor s' * r \rfloor \subseteq \lfloor t' * r \rfloor)$$

We must show it satisfies the conditions of definition 17. In ④ , pick $\alpha = \textsc{id}$, $s = p$ and $t = p$. Expanding, we derive:

$$(\forall r \in \textsc{Views}.\ \lfloor p * r \rfloor \subseteq \lfloor p * r \rfloor) \implies$$
$$(\forall r, s', t' \in \textsc{Views}.\ p \sim s' \wedge p \sim t' \implies \lfloor s' * r \rfloor \subseteq \lfloor t' * r \rfloor)$$

The first term of the first implication is a tautology, and so the entire assertion is equivalent to:

$$(\forall r, s', t' \in \textsc{Views}.\ p \sim s' \wedge p \sim t' \implies \lfloor s' * r \rfloor \subseteq \lfloor t' * r \rfloor)$$

Now pick $s' = p$ and $t' = q$. Expanding, we derive:

$$(\forall r \in \textsc{Views}.\ p \sim p \wedge p \sim q \implies \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor)$$

The first conjunction is discharged by ② (reflexivity). The second contains no mention of r. Ergo:

$$p \sim q \implies \forall r \in \textsc{Views}.\ \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$$

which was to be shown. □

That the rules are interchangeable with reflexive and symmetric relations is perhaps not surprising. Take the program $\mathbb{C}$ and views $p^\top, q^\top, p, q$. Imagine a valid derivation, a fragment of which has the form below:

$$\dfrac{\dfrac{\vdots}{\{p^\top\}\ \mathbb{C}\ \{q^\top\}}}{\{p\}\ \mathbb{C}\ \{q\}}$$

We can see the centre triple as partaking in forms of *strengthening* or *weakening* on p, depending on if we read it bottom to top, or top to bottom (the arguments equally apply to q). Reading bottom to top, we see p as getting weaker; $p^\top$ must encompass p. Reading top to bottom, we see p as getting stronger; $p^\top$ is more expansive than p

Without the frame rule, generalised frame rule, or relational frame rule, the strengthening reading is trivial; $p^\top = p$ is the only valid choice. However, with the various frame

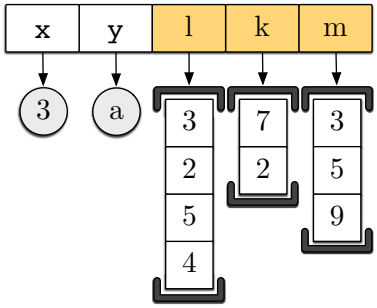


Figure 3.19.: Structured heap containing multiple lists.

rules, our choices for strengthen the pre-condition are much wider. The standard frame rule strengthens p with more memory cells, justified via the inability of the machine to detect memory locations not accessed. These rules are not well explored; the views framework [22] includes a penalised frame function that expresses the weakening rule used by Rely-Guarantee [47].

For structural separation logic, abstract allocation is used only to change the presentation of the data. We make certain sub-data more accessible, but never change the meaning of heap from the machines perspective. As we are neither strengthening or weakening our view, either rule is equally applicable for abstract allocation. We choose to use semantic consequence, as it can be used both for abstract allocation *and* the folding and unfolding of inductive predicates in separation logic. As we will be using both compression/decompression and inductive predicates in our examples, the uniformity of style is useful.

3.5.2. Disjoint structural addresses

We now consider reasoning for libraries that need knowledge of the *machine heap address* that is storing their data. In our structured heap abstraction, such situations arise often. When multiple copies of a data structure are allowed, the heap address can be used to identify the specific *instance* being analysed. Recall the list example of section 3.1.1. This works with exactly one list, so that the programmer to create multiple lists in the heap. When faced with the assertion $\alpha \mapsto 5$, there is no ambiguity about which list contained 5. The command `remove(5)` would remove the only element 5 there could possibly be.

This is unrealistic. It is quite reasonable to have many lists in the heap, each of which may have an element 5. These lists are created, destroyed and updated as needed with program commands. Typically, lists are associated with a unique address, and commands

are parameterised with the address of the list being accessed.

We can build an imperative machine for *multiple lists* using the list data structures we have already defined (definition 26). The machine heaps to support this are similar to those of the single list case, but contain zero or more lists.

Definition 66 (Multiple-list machine heaps)**.** Assume a set of list heap addresses ListAddrs, ranged over by $l, k, m, \cdots$, where $\textsc{ListAddrs} \cap \textsc{PVars} = \emptyset$. The set of **multiple-list machine heaps** ListHeaps, ranged over by $lh, lh_1, \cdots, lh_n$, is:

$$\textsc{ListHeaps} \triangleq \{lh \mid lh : (\textsc{ListAddrs} \xrightarrow{\text{fin}} \textsc{Lists}) \sqcup (\textsc{PVars} \xrightarrow{\text{fin}} \textsc{PVals})\}$$

We visualise such a heap, with two variables `x` and `y` and three lists $a \mapsto [3 \otimes 2 \otimes 5 \otimes 4]$, $d \mapsto [7 \otimes 2]$, and $b \mapsto [3 \otimes 5 \otimes 9]$ in figure 3.19. Note that whilst individual lists must have distinct elements, different lists may have the same elements. To manipulate these heaps, we alter the list library given in chapter 3 to support multiple lists:

1. `l := createList()`: Picks a machine fresh heap address, allocating it within the heap, associating it with an empty list, and assigning it to variable `l`. Cannot fault.
2. `destroyList(l)`: Removes the entire list identified by the value of `l` from the heap. Faults if the value of `l` does not identify a list.
3. `l.append(`e`)`: Appends the result of evaluating e to the list identified by the value of `l`, making it the last element. Faults if `l` does not identify a list, if the evaluation of e is either already in that list, or if the evaluation of e is not a natural number.
4. `l.remove(`e`)`: Removes the result of evaluating e from the list identified by the value of `l`. Faults if `l` does not identify a list, or if the evaluation of e is not present in that list.

For brevity, we omit the similar commands for `l.getFirst(`e`)` and `l.getRight(`e`)`; they follow this pattern.

The construction of abstract heaps over these machine heaps is not, by itself, useful. Consider the assertion $\alpha \mapsto 3$. It gives no information about the list to which element 3 belongs. Instead, we consider associating each abstract address with the address of the list into which it will ultimately compress:

$$\alpha^{\text{L}} \mapsto \text{V}$$

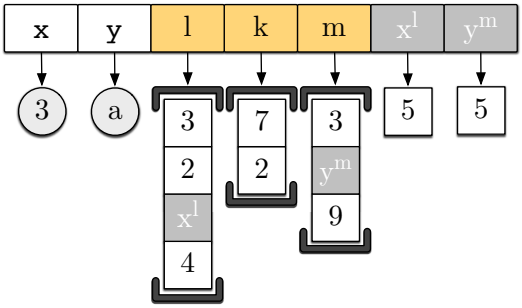


Figure 3.20.: Abstract heap using tagged addresses to manage multiple lists.

This abstract cell assertion states that address α was allocated from within the list L. With it, we can axiomatise the commands for working with multiple lists.

Formalisation

We call this approach **tagged structural addresses**, and formalise it for data in general, later focusing it on the list specific case. Given a set of structural addresses, we create new sets, each containing structural addresses *tagged* with a machine address. Tagged structural addresses are constructed atop the standard structural addresses:

Definition 67 (Tagged structural addresses)**.** Let $a \in \textsc{MachineAddrs}$ be a machine address. The set of **structural addresses tagged with** a, ranged over by $\mathbf{x}^a, \mathbf{y}^a, \mathbf{z}^a$, is defined as:

$$\textsc{StructAddrs}^a = \{\mathbf{x}^a \mid \mathbf{x} \in \textsc{StructAddrs}\}$$

From a structural addressing algebra $(\textsc{StructAddrs}, \textsc{Data}, \mathrm{addrs}, \mathrm{comp})$, we create new algebras for each machine address, using the tagged structural addresses $(\textsc{StructAddrs}^a, \textsc{Data}^a, \mathrm{addrs}^a, \mathrm{comp}^a)$. This construction is straight-forward, simply substituting the corresponding tagged address for each use of a structural address. When building the abstract heap, the data stored at a machine address will always use the set of data tagged with appropriate machine address, as seen in figure 3.20. Such abstract heaps are:

Definition 68 (Abstract heaps over tagged data)**.** The set of **abstract heaps using tagged data and addresses** AbsHeaps is defined as:

$$\textsc{AbsHeaps} \subseteq \left\{ \mathbf{h} \;\middle|\; \mathbf{h} : \bigsqcup_{a \in \textsc{MachineAddrs}} \begin{pmatrix} (\textsc{PVars} \xrightarrow{\text{fin}} \textsc{PVals}) \\ \sqcup \\ \{a\} \rightharpoonup \textsc{Data}^a \\ \sqcup \\ \textsc{StructAddrs}^a \xrightarrow{\text{fin}} \textsc{Data}^a \end{pmatrix} \right\}$$

By construction, these abstract heaps require that body addresses within data must be tagged with the machine address into which the data will collapse. As we are simply changing the set of abstract addresses, we require no new machinery for the collapse process or reification. The normal compression function will ensure that the collapse relation can only place tagged abstract addresses into the appropriate places. The completion process will then ensure that the needed additional cells are provided.

The assertion language is essential identical to the single machine address case. We update the logical environments to ensure that any *tagged logical variable* refers to an underlying structural address that shares the tag. This prevents a logical variable α^a from actually storing a structural address $\mathbf{x}^b$, where $a \neq b$.

Definition 69 (Logical environment)**.** Given a logical environment $\Gamma \in \textsc{LEnvs}$ (definition 20), the *tagged logical environment* $\textsc{LEnvs}_T : \textsc{LVars} \rightharpoonup \textsc{LVals}$ is defined as: for all $\alpha, a, \mathbf{x}, b$

$$\Gamma_T(\mathrm{x}) = \begin{cases} \mathbf{x}^a & \text{if } \mathrm{x} = \alpha^a, \Gamma(\alpha^a) = \mathbf{x}^a \\ \text{undefined} & \text{if } \mathrm{x} = \alpha^a, \Gamma(\alpha^a) = \mathbf{x}^b, a \neq b \\ \Gamma(\mathrm{x}) & \text{otherwise} \end{cases}$$

We assume the use the tagged logical environment rather than a normal logical environnent through all definitions. Abstract allocation then requires a small change to use these tagged logical variables, but no new underlying theory. This is because, than being a definition, abstract allocation is a result of the semantic consequence relation 17. This was by careful design of the abstract heaps and reification function, which ensured that as long as the *collapse* of two heaps resulted in identical machine heaps, they were semantic

consequences of each other. We have the same property here. By design, these tagged structural addresses can be used to compress and de-compress data as before, as long as the correct addresses are chosen for the data.

We can therefore easily update the relation to use the set of structural addresses appropriate to the data being allocated. This can always be determined by looking at the address of the heap cell the sub-data is being allocated from.

Proposition 4 (Abstract allocation on tagged addresses)**.** The following semantic equivalence is valid: for all $a \in \textsc{MachineAddrs}$

$$\left(a \mapsto \phi_1 \textcircled{\alpha^a} \phi_2\right) \preccurlyeq\succcurlyeq_\Gamma \exists \alpha^a.\, (a \mapsto (\phi_1 \wedge \diamond\alpha^a) * \alpha^a \mapsto \phi_2)$$
$$\left(\alpha^a \mapsto \phi_1 \textcircled{\beta^a} \phi_2\right) \preccurlyeq\succcurlyeq_\Gamma \exists \beta^a.\, (\alpha^a \mapsto (\phi_1 \wedge \diamond\beta^a) * \beta^a \mapsto \phi_2)$$

The proof that this proposition holds is similar to that of theorem 4.

Tagged structural addressing for multiple lists

By creating tagged data and tagged addressing algebras using $\textsc{ListAddrs}$, we obtain abstract heaps for multiple lists. With these heaps, we can construct the axioms for the multiple list commands, which are given in figure 3.21. The creation and destruction commands are very similar to those for normal heap cells. The `append` command works on an entire list cell, the address of which is passed as a parameter. The `remove` is the most changed, using an abstract heap cell with a tagged address to indicate that α is allocated from data stored in the $\textsc{l}$ machine cell. Abstract allocation then behaves in the intuitive manner. In the list instance, the following two semantic consequences hold:

$$\textsc{l} \mapsto [1 \otimes 2 \otimes 3] \preccurlyeq\succcurlyeq_\Gamma \exists \alpha^{\textsc{l}}.(\textsc{l} \mapsto [1 \otimes \alpha^{\textsc{l}}] * \alpha^{\textsc{l}} \mapsto 2 \otimes 3)$$
$$\alpha^{\textsc{l}} \mapsto 2 \otimes 3 \preccurlyeq\succcurlyeq_\Gamma \exists \beta^{\textsc{l}}.(\alpha^{\textsc{l}} \mapsto 2 \otimes \beta^{\textsc{l}} * \beta^{\textsc{l}} \mapsto 3)$$

Notice that, out of an abstract cell addressed by $\alpha^{\textsc{l}}$, one can only allocated new abstract cells of with addresses of the form $\beta^{\textsc{l}}$. This is because, by construction, such a cell is storing data where the only body addresses are tagged with $\textsc{l}$.

3.6. Summary

This chapter has introduced **structural separation logic**, a program reasoning system for libraries manipulating structured data. Structural separation logic consists of two key contributions:

$$\{\mathtt{l} \to \text{-}\}$$
$$\mathtt{l} := \mathtt{createList}()$$
$$\{\exists \mathrm{L}.\ \mathrm{L} \mapsto [\varnothing] * \mathtt{l} \to \mathrm{L}\}$$

$$\{\mathrm{L} \mapsto [\mathrm{E} \wedge \mathit{is\_complete}] * \mathtt{l} \to \mathrm{L}\}$$
$$\mathtt{destroyList}(\mathtt{l})$$
$$\{\mathtt{l} \to \mathrm{L}\}$$

$$\{\mathrm{L} \mapsto [\mathrm{E} \wedge \mathit{is\_complete}] * \mathtt{l} \to \mathrm{L} * P * e \Rightarrow \mathrm{R} \wedge \mathrm{R} \notin \mathrm{E}\}$$
$$\mathtt{l.append}(e)$$
$$\{\mathrm{L} \mapsto [\mathrm{E} \otimes \mathrm{R}] * \mathtt{l} \to \mathrm{L} * P\}$$

$$\{\alpha^{\mathrm{L}} \mapsto \mathrm{I} * \mathtt{l} \to \mathrm{L} * P \wedge e \Rightarrow \mathrm{I}\}$$
$$\mathtt{l.remove}(e)$$
$$\{\alpha^{\mathrm{L}} \mapsto \varnothing * \mathtt{l} \to \mathrm{L} * P\}$$

Figure 3.21.: Example small axioms for the multiple list library

- A model with abstract separation: Section 3.4 defines an algebra for structured data (definition 42) that, when combined with abstract heaps (definition 48), gives a model for splitting entirely abstract data into bags of sub-data that can be considered independently, then reconnected.
- A logic with abstract allocation: Section 3.3 shows how this model can be used as the basis of a separation logic, with splittings managed via *abstract allocation*.

As discussed in section 3.4, these enable several advances of previous techniques for local abstract data reasoning.

1. **Abstraction of separability**: Structural separation logic offers an *abstraction of separability* (or perhaps *connectivity*) for structured data. Most clearly demonstrated in section 3.4, this allows a greater level of abstraction for data than previous separation logic techniques utilising abstract predicates.
2. **General approach for abstract data**: Section 3.4 provides a uniform approach for separation logic reasoning with abstract structured data. As demonstrated in section 3.4, our use of separating conjunction offers advantages over the previous work in context logic, enabling both smaller axioms and concurrent reasoning.
3. **Coexistence with existing separation logic techniques**: Unlike previous work in this area, the abstract heap concept of structural separation logic can reason about programs that manipulate both high-level abstract data and low-level heap

structures. This will prove valuable in the upcoming case studies of chapter 4 and 6.

4. The Document Object Model

This chapter uses structural separation logic to present an axiomatic specification for the W3C *Document Object Model* library standard [71]. The Document Object Model (hereafter DOM) library is typically used to analyse and update HTML or XML documents. The standard provides an abstract representation of the trees, and describes a wide range of commands for manipulating these trees. The most common use of DOM is manipulating the content of web pages. Every modern web browser comes with an implementation of DOM that should adhere to the standard, and this implementation is exposed to script programs within a page. This allows code delivered with a web page to alter the structure of the tree representing the browsers internal view of the HTML, and hence the presentation of the web page.

As a library, DOM is exclusively focused on the manipulation of structured data. In figure 4.1, we see the DOM tree generated by parsing a simple HTML document. It consists of a set of *nodes* of varying types. The DOM standard describes the types of nodes and their relationships, and gives a comprehensive set of commands for manipulating them. These commands are similar to those of our tree update library (section 3.1), but are much more extensive.

DOM is very widely adopted. For one, every web browser contains an implementation. That implementations follow the standard is critical, the same script will be delivered to many different web browsers, and script authors expect the behavior to be the same in each case. However, DOM is is also used outside of browsers, for creating and processing XML files. XML has become a commonly used standard for data interchange, and most platforms provide a DOM library. Giving an axiomatic specification to the DOM library gives an unambiguous semantics to a widely used specification, a valuable outcome.

DOM is an attractive example for formalisation with structural separation logic. It takes the essence of our small tree examples, and scales it to a real-world problem, allowing us to show our reasoning scales as well. Moreover, it enables reaosning about usages of the DOM library inside other local reasoning logics. Smith, Gardner, and Maffeis are currently extending their separation logic reasoning for JavaScript [34] towards a more complete version of the language, capable of reasoning about real web programs. To

```
<html>
  <head>
    <title>Adam's page</title>
  </head>
  <body>
    Lorum ipsum…
  </body>
</html>
```

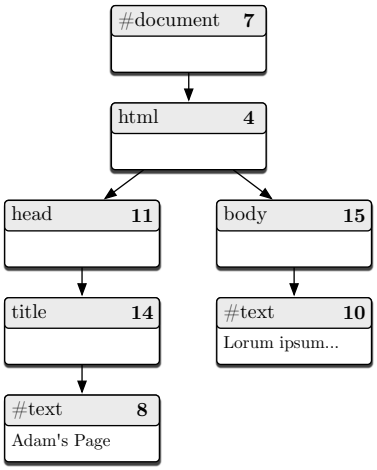


Figure 4.1.: Parsing HTML or XML with a DOM parser results in a *DOM tree*. Each node is uniquely identified, and there are three node types: a *#document* node, which is the root of the tree, *element* nodes which have a name and children, and *#text* nodes, which have associated text.

ensure soundness of these programs, they will require separation logic reasoning for the DOM, something we can now provide.

DOM has previously been studied in the context of local reasoning. As the standard is very large, Gardner, Wheelhouse, Smith and Zarfaty identified a subset they called *featherweight DOM*. We will base our analysis on the same subset, and demonstrate that structural separation logic can provide significantly smaller footprints than were possible with context logic. We will also show the advantages we gain via close integration with heap reasoning, by creating an example that uses both the DOM library and normal heap manipulation.

This chapter follows the pattern set out for the simple tree library given in section 3.1. In section 4.1, we introduce a data structure and structured heap for representing DOM, using the structures introduced by featherweight DOM [35] as the starting point. In section 4.1.1, we introduce the DOM commands, adopted from featherweight DOM, and give operational actions to four of them. Together, these objects provide an imperative machine for programming with the DOM library. We give examples programs for this machine by implementing additional commands not present in the featherweight subset.

In section 4.2.1, we define a structural addressing algebra (definition 42) for DOM, and so give the abstract heaps (definition 48). These enables us to use structural separation logic to reasoning about DOM programs. We axiomatise the featherweight DOM command

set, and prove the examples given in the previous section. In creating our axiomatisation, we uncovered that featherweight DOM slightly deviates from the English specification. We identify why and demonstrate that we have corrected the oversight.

This chapter concludes in section 4.4 with an example of *hybrid* reasoning. We give an program that uses normal heap operations alongside the DOM library, in the form of a *photo library processor*. We consider a photographic library stored in an XML file. The file contains a list of photos, each associated with a name and some image data. As XML is a plain text format, and the image data is an arbitrary list of bytes, an *encoding* is used to represent the image in the XML. Our example program uses the DOM to iterate over the images in the file. It must decode the image data into a standard format, but as the images are all of different sizes, it is standard to use heap memory to store the decoded data. Proving the example memory safe requires a combination of proof techniques, using the DOM reasoning to verify that the XML data is used in a proper manner, and standard heap reasoning to ensure the memory containing the image is used correctly.

4.1. DOM library

There are currently three major versions of the DOM standard. Like [35], we focus our reasoning on a fragment of *DOM Core Level 1*, which defines the general shape of the trees and majority of commands for manipulating them. Later versions are mostly concerned with *event handling* and minor updates to the tree shape. We will not work with the entirety of the standard, instead using a small variant on the featherweight interface identified in [35] that extends the subset with *document nodes*. By using this fragment, we focus on the core difficulties of the reasoning without handling the verbosity of the entire document. This fragment was extended to the entire standard by Smith [65]. We expect a similar extension of our work to be equally possible.

DOM Core Level 1 describes the library in object-oriented manner, defining both a data structure and operations for manipulating it via a set of *interfaces*. Acknowledging that not all languages support object-orientation, the library also provides *"a simplified view that allows all manipulation to be done via the* `Node` *interface"*. We will base our tree data structure upon the relationships between the objects in the object-oriented interfaces, and base our commands upon the single `Node` interface.

The standard defines a *DOM document* as a collection of *nodes* related by structural invariants. Each of the node types has some common structure:

Name: Every node has a name, not necessarily distinct across a document.

Node identifier: Every node has an identifier, which is distinct across a document.

Associated data: Each node has some associated data, the type of which depends on the node type.

Node lists: Every node is associated with a set of *forest identifiers*. When asked, the node must provide an element of this set, which will be used to obtain any children the node may have. The standard mandates that queries for children are always against the *live* state of a node, so it is impossible to simply return a static snapshot of children. At each query for a forest identifier, it is unknown whether an element already in the set will be returned, or whether the set will extended with fresh identifiers, and one of those returned. This non-determinism is due to under-specification in the standard.

DOM Core level 1 provides twelve node types, of which we consider the three most commonly used. These three are sufficient to demonstrate the reasoning, and write effective examples. The additional node types are similar, and can be added to the presentation with minimal difficulty. The three node types we model are:

Element nodes: Element nodes are the most common DOM node, representing a named node with children. They have arbitrary names, with the caveat that the name cannot contain the character `#`, which is reserved as a prefix of for the names of other node types. The data of these nodes is an ordered list of *child nodes*, which we call the *child forest*. Our basic notation for elements is:

$$\text{Name}_{\text{Identifier}}[\text{Child Forest}]_{\text{Set of forest identifiers}}$$

Text nodes: Text nodes represent blocks of text, such as `This is a text node's contents`. These are always named `#text`. The data is a possibly empty string. Although text nodes have no children, they still have a set of forest identifiers. Whenever these identifiers are used to query for a child, the query will report that there are no children. Our basic notation for text nodes is:

$$\#\text{text}_{\text{Identifier}}[\text{String}]_{\text{Set of forest identifiers}}$$

Document nodes: Each document has exactly one document node, which is always the root of the DOM tree. They are named `#document`, and contain at exactly one associated child node, called the *document element*. The set of forest identifiers may seem redundant (there is only ever one element), but is required by the DOM standard.

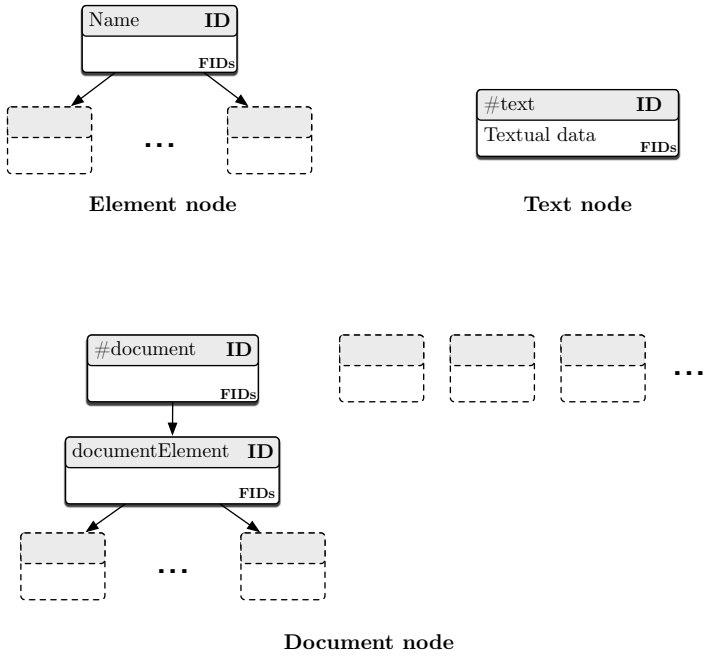


Figure 4.2.: The three node types we model in our DOM structure. The nodes with dashed outlines represent nodes that are optional.

Alongside the document node, we store the set of orphaned elements and text nodes, which we call the *grove*[1]. The grove is where newly created nodes will be placed. Also, DOM has no commands to dispose a node. Whenever a node is no longer needed, it is returned to the grove and "forgotten about". It is thus natural to consider DOM as garbage collected. Our basic notation for document nodes is:

$$\#\text{document}_{\text{Identifier}}[\text{Document element}]_{\text{Set of forest identifiers}} \;\&\; \text{Grove node set}$$

The node types are rendered in figure 4.2. The relationships between the node types defines the structure of a *DOM document*. Each document always has exactly one document node, acting as the root. This document node has exactly one *element* node child. This *document element* node is a normal element, and may have zero or more children. Text nodes can only appear at the leaves of the tree. Orphaned nodes are placed within the grove, but are not considered roots of the DOM tree. An example of this structure is given in figure 4.3.

[1] As we are working with a minor extension to featherweight DOM that includes document nodes, our choice of grove definition is slightly different from previous work [35].

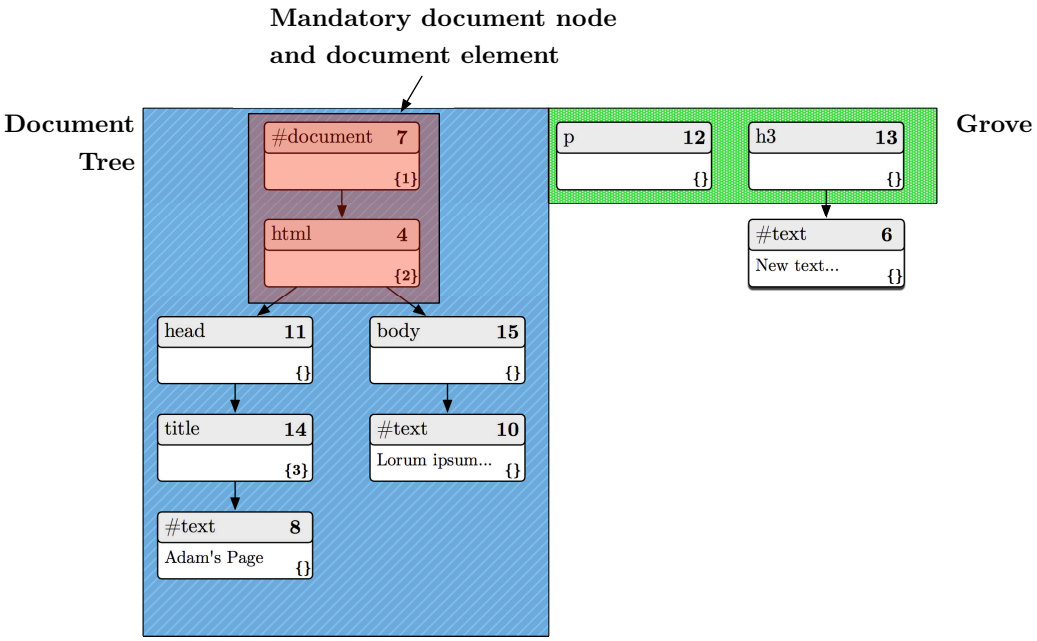


Figure 4.3.: Example structure of a DOM document. The shading indicates the key structured of a DOM tree. Notice that some nodes have empty forest identifier sets. This is a valid choice, assuming those nodes have not yet been queried for a forest identifier.

4.1.1. An imperative machine for DOM

We now build an imperative machine for DOM, formalising the document structure and command subset. We use the following objects to represent node identifiers, forest identifiers, the null (that is, invalid) node identifier, and a set of characters suitable for the strings of DOM.

Parameter 18 (Primitive DOM sets)**.** Assume the following objects:

- A countably infinite set of **node identifiers** NodeIDs, ranged over by $n, n_1, \cdots, n_n, m, o$.
- A countably infinite set of **forest identifiers** ForestIDs, ranged over by $fid, fid_1, \cdots, fid_n$, such that ForestIDs$\cap$NodeIDs $= \emptyset$. Subsets of ForestIDs are ranged over $fs, fs_1, \cdots, fs_n$.

- A set of **DOM characters** DOMChars, ranged over by $c, c_1, \cdots, c_n$.
- A distinguished value **null** such that **null** $\notin$ NodeIDs $\cup$ ForestIDs.

The data structures representing DOM trees are defined as follows.

Definition 70 (DOM data structure)**.** Given the primitive DOM sets of parameter 18, the set of **DOM strings** DOMStrings, ranged over by $s, s_1, \cdots, s_n$, the set of **DOM trees** DOMTrees, ranged over by $t, t_1, \cdots, t_n$, the set of **DOM forests** DOMForests, ranged over by $f, f_1, \cdots, f_n$, the set of **DOM groves** DOMGroves, ranged over by $g, g_1, \cdots, g_n$, and the set of **DOM documents** DOMDocs, ranged over by $doc, doc_1, \cdots, doc_n$, are defined by induction as follows: for all $c \in$ DOMChars, $n \in$ NodeIDs and $fs \in \mathcal{P}($ForestIDs$)$

$s ::=$	$c \mid s_1 \cdot s_2 \mid \varnothing_s$	DOM strings
$t ::=$	$s_n[f]_{fs} \mid \#\text{text}_n[s]_{fs}$	DOM trees
$f ::=$	$t \mid f_1 \otimes f_2 \mid \varnothing_f$	DOM forests
$g ::=$	$t \mid g_1 \oplus g_2 \mid \varnothing_g$	DOM groves
$doc ::=$	$\#\text{document}_n[t]_{fs}$ & g	DOM documents, t an element node

where $\cdot$ is associative with left and right identity $\varnothing_s$, $\otimes$ is associative with left and right identity $\varnothing_f$, and $\oplus$ is associative and commutative with identity $\varnothing_g$. All data are equal up to the properties of $\cdot, \otimes$ and $\oplus$, and contain no duplicate node or forest identifiers.

It is often useful to refer to "DOM data" in general, without knowing whether it is a string, element, text node, grove or document.

Definition 71 (DOM data)**.** The set of **DOM data** DOMData, ranged over by $d, d_1, \cdots, d_n$, is the union of the DOM data types:

$$\text{DOMData} \triangleq \text{DOMStrings} \cup \text{DOMTrees} \cup \text{DOMForests} \cup \text{DOMGroves} \cup \text{DOMDocs}$$

By definition 70, DOM strings are lists of DOM characters. DOM trees are either an *element* node (with a name, identifier, child forest and forest identifier) or a *text node*

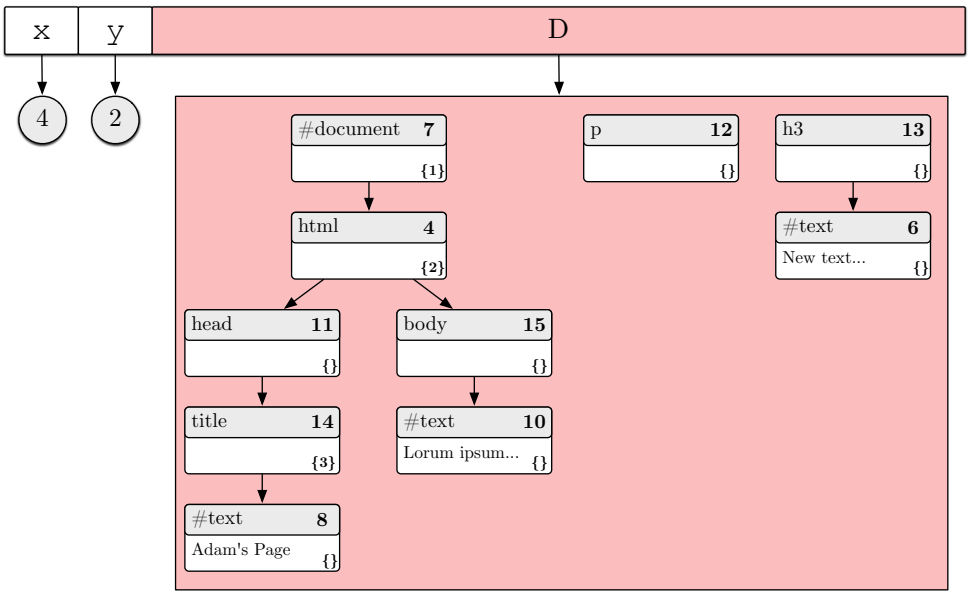


Figure 4.4.: DOM heap structure

(with an identifier and child string). DOM forests are possibly empty lists of DOM trees. DOM groves are possibly empty *sets* of DOM trees. DOM documents are a *document node* with identifier and forest identifier, but have exactly one child element node (which we call the *document element*), and are associated with a grove. This definition is a minor variant on that of featherweight DOM [35].

We create structured heaps for DOM exactly as we did for simple lists and trees in section 3.1.

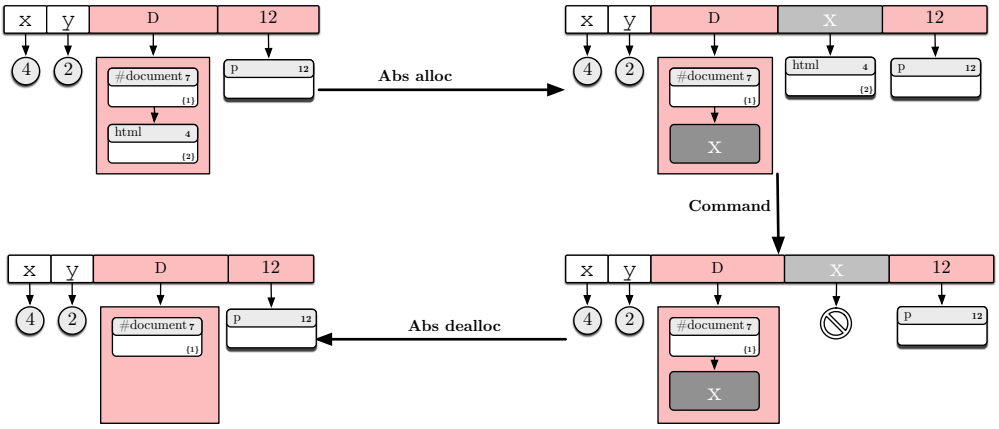
Definition 72 (DOM structured heaps)**.** Given the DOM sets (parameter 18) and the DOM documents DOMDocs (definition 70), and letting the program variables $\text{PVals} = \text{NodeIDs} \cup \text{ForestIDs} \cup \text{DOMStrings} \cup \{\textbf{null}\} \cup \mathbb{Z} \cup \mathbb{B}$ and the DOM tree address be $\text{D} \notin \text{PVars}$, the set of **DOM structured heaps**, ranged over by $ds, ds_1, \cdots, ds_n$, have the type:

$$\text{StructHeaps} \subseteq (\text{PVars} \xrightarrow{\text{fin}} \text{PVals}) \sqcup (\{\text{D}\} \to \text{DOMDocs})$$

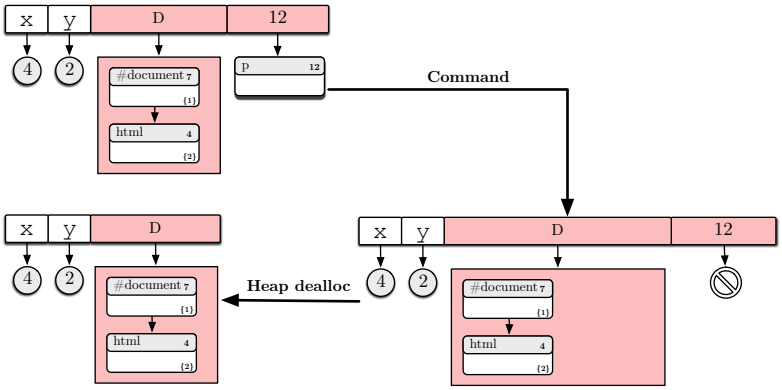
To simplify the presentation, there is exactly one DOM tree in the heap. This models the typical use of DOM in a web browser, where a single tree modelling the currently active page is provided. This can be extended to multiple DOM trees using the tagged

structural addresses of 3.5.2.

In the heap, there is a delineation between the set of heap addresses, and the set of DOM identifiers within the heaps. We do not use DOM identifiers as a heap address at any point. The DOM tree is confined to the value of address D (see figure 4.4). It is, in principle, possible to store DOM data directly in the heap, with #document and grove node identifiers being heap addresses. This choice, however, does not treat nodes uniformly, as is the spirit of DOM. It also comes at the price of uniformity in the reasoning. Consider such a scenario, where grove nodes are stored directly in the heap. For example, a command that removed a non-grove node would use abstract allocation to isolate the node, and behave as:



The same command, when applied to a grove node, would behave as:



Ergo, the command would have to account for either a "deep" use of the data (an abstract heap cell), or a "shallow" use of the data (a grove cell). In the first, the cell is removed by abstract deallocation. In the second, an actual heap deallocation step would be required. By isolating all DOM nodes inside the DOM tree cell at address D, and reasoning about them using only abstract heap cells, we ensure uniformity.

Commands

The commands of our DOM machine are those of the variable system (example 2), plus the DOM commands of featherweight DOM. These commands were chosen as a subset of DOM that demonstrates the range of structural behaviours. Moreover, they were chosen to be *minimal* and *sufficient*, in that they can be used to implement many additional standard DOM commands. We will give some implemented commands in section 4.1.2. Informally, given a DOM structured heap, the commands behave as follows:

`n := createElement(`e`)` : If the expression e evaluates to a DOM string s, allocates a fresh node identifier n and set of forest identifiers fs, adds $s_n[\varnothing_f]_{fs}$ to the end of the grove of the DOM document, and assigns n to `n`. Faults if e does not evaluate to a DOM string, or if the string contains the character #.

`n := createTextNode(`e`)` : If the expression e evaluates to a DOM string s, allocates a fresh node identifier n and set of forest identifiers fs, adds $\#\text{text}_n[s]_{fs}$ to the end of the grove of the document, and assigns n to `n`. Faults if e does not evaluate to a DOM string.

`s := n.nodeName` : Assigns to `s` the name of the node structure identified by variable `n`. Faults if `n` does not map to a node identifier, or if `n` does not identify a node in the document.

`m := n.parentNode` : Assigns to `m` the node identifier of the parent of the node identified by `n`. If `n` identifies a document node, or if it identifies a node in the document grove, assigns **null**. Faults if `n` does not identify a node.

`c := n.childNodes` : If `n` identifies a node n, adds zero or more fresh forest identifiers to the forest identifier associated with n, then assigns an element of the resultant set to `c`.

`n := f.item(`e`)` : If the expression $e$evaluates to an integer i, and `f` maps to a forest identifier present in the set associated with some node n, returns the $i + 1^{\text{th}}$ node of

the child forest of n, assigning it to `n`. If i is less than zero, or greater than or equal to the length of the child forest, assigns **null**. Faults if the evaluation of e faults, or if `f` does not map to a forest identifier associated with some node.

`o := n.removeChild(m)` : Removes the node identified by `m` from the child forest of the element node identified by `n`, placing it at the end of the grove list of the document, and assigning the value of `m` to `o`. Faults if either `n` or `m` do not identify nodes, if `n` identifies a document or text node, or if the node identified by `m` is not a child of that identified by `n`.

`o := n.appendChild(m)` : If `n` identifies an element node n and `m` identifies a element or text node m that is not an ancestor of n, removes node m from any parent it may have, places it at the end of the child forest of node n, and assigns m to the variable `o`. Faults if `n` or `m` do not identify nodes, if `m` if an ancestor of `n`, if `m` identifies a document node, or if `n` identifies a text node or a document node with an existing document element.

`s := n.substringData(`e_1`,` e_2`)` : If `n` identifies a text node with associated data string s, the expressions e_1 and e_2 evaluate to integer values i and j respectively, where $i \geq 0$ and $j \geq 0$, then assigns the sub-string of s beginning at index i and continuing for j characters to `s`. If i is greater than the length of the string, assigns the empty string. If $i + j$ exceeds the length of the string, assign the sub-string from index i to the end of the string. Faults if `n` does not identify a text node, if the evaluation of either e_1 or e_2 faults, or if i or j are negative.

`n.appendData(`e`)` : If `n` maps to the identifier of some text node, and $e$evaluates to some DOM string s, appends s to the child string of the node identified by `n`. Faults if `n` does not identify a text node, if the evaluation of e faults, or if s is not a DOM string.

`n.deleteData(i, j)` : Behaves as `n.substringData(i j)` but rather than returning the data, removes it from the string.

In all the commands, we use expressions for non-identifier parameters, but just variables for identifiers. There are no useful expressions for identifiers other than simple variable lookup, as they are all generated at runtime non-deterministically (so have no useful literals), and there are no expression operations on them. Using just variables here allows us to give simpler axioms.

The commands break down into four groups. The *creation* commands `createElement` and `createTextNode` create new element and text nodes, placing them at the grove level. These commands are straightforward. Notice there is no command to create a document node, as the heaps we work with must contain exactly one. The *query* commands `nodeName` `parentNode`, `childNode` and `item` extract properties of a node given its node identifier. Most are straightforward, but the behaviour of `n.childNodes` is unusual. Rather than returning a fixed forest identifier, it behaves non-deterministically. Before returning a forest identifier from the set associated with node `n`, it first adds zero or more fresh identifiers to the set. It therefore returns either an existing identifier or a new identifier. The behaviour is necessary to account for an under-specification in the DOM standard, which we investigate further in section 4.3.

The *structural* commands `appendChild` and `removeChild` are standard. Note that `removeChild` does not destroy the node and the sub-tree, but places it at the grove level. Finally, the *string* commands analyse and update the contents of text nodes. They operate only on the contents of text nodes, rather than on DOM strings in general.

We formalise only a representative subset of of these commands into actions. The number of cases to consider for even the representation set is large, and the rest behave as would be expected from their informal definitions above. For lists and trees, we created the actions using partial lists and single-holed tree contexts respectively (sections 3.1.1 and 3.1.2). Here, we will define the abstract DOM data and compression function that we will use for the abstract heaps, making use of these to define the actions.

Definition 73 (Abstract DOM data)**.** The set of **abstract DOM strings** AbsDOMStrings, ranged over by $\mathbf{s}, \mathbf{s}_1, \cdots, \mathbf{s}_n$, the set of **abstract DOM trees** AbsDOMTrees, ranged over by $\mathbf{t}, \mathbf{t}_1, \cdots, \mathbf{t}_n$, the set of **abstract DOM forests** AbsDOMForests, ranged over by $\mathbf{f}, \mathbf{f}_1, \cdots, \mathbf{f}_n$, the set of **abstract DOM groves** AbsDOMGroves, ranged over by $\mathbf{g}, \mathbf{g}_1, \cdots, \mathbf{g}_n$, and the set of **abstract DOM documents** AbsDOMDocs, ranged over by $\mathbf{doc}, \mathbf{doc}_1, \cdots, \mathbf{doc}_n$, are defined by induction as follows: for all $c \in$ DOMChars, $n \in$ NodeIDs and $fs \in \mathcal{P}($ForestIDs$)$, $\mathbf{x} \in$ StructAddrs

$$
\begin{array}{lll}
\mathbf{s} ::= & c \mid \mathbf{s}_1 \cdot \mathbf{s}_2 \mid \varnothing_s \mid \mathbf{x} & \text{Abstract DOM strings} \\
\mathbf{t} ::= & s_n[\mathbf{f}]_{fs} \mid \#\text{text}_n[\mathbf{s}]_{fs} & \text{Abstract DOM trees} \\
\mathbf{f} ::= & \mathbf{t} \mid \mathbf{f}_1 \otimes \mathbf{f}_2 \mid \varnothing_f \mid \mathbf{x} & \text{Abstract DOM forests} \\
\mathbf{g} ::= & \mathbf{t} \mid \mathbf{g}_1 \oplus \mathbf{g}_2 \mid \varnothing_g \mid \mathbf{x} & \text{Abstract DOM groves} \\
\mathbf{doc} ::= & \#\text{document}_n[\mathbf{t}]_{fs} \ \& \ \mathbf{g} & \text{Abstract DOM documents} \\
& \mid \#\text{document}_n[\mathbf{x}]_{fs} \ \& \ \mathbf{g} & \mathbf{t} \text{ with an abstract element node}
\end{array}
$$

where $\cdot$ is associative with left and right identity $\varnothing_s$, $\otimes$ is associative with left and right identity $\varnothing_f$, and $\oplus$ is associative and commutative with identity $\varnothing_g$. All data are equal up to the properties of $\cdot, \otimes$ and $\oplus$, and contain no repeated node or forest identifiers, nor repeated structural addresses.

The set AbsDOMData, ranged over by $\mathbf{d}, \mathbf{d}_1, \ldots, \mathbf{d}_n$, is the union of all these sets:

$$
\text{AbsDOMData} = \begin{array}{c} \text{AbsDOMStrings} \cup \text{AbsDOMTrees} \cup \text{AbsDOMForests} \\ \cup \text{AbsDOMGroves} \cup \text{AbsDOMDocs} \end{array}
$$

Definition 74 (Compression for DOM)**.** Given a set of structural addresses StructAddrs (definition 35) and abstract DOM data AbsDOMData (definition 73), the compression function for the DOM structure comp : StructAddrs $\to$ AbsDOMData $\to$ AbsDOMData $\rightharpoonup$ AbsDOMData is defined as substitution, $\text{comp}(\mathbf{x}, \mathbf{d}_1, \mathbf{d}_2) = \mathbf{d}_1[\mathbf{d}_2/\mathbf{x}]$. An application $\text{comp}(\mathbf{x}, \mathbf{d}_1, \mathbf{d}_2)$ is written $\mathbf{d}_1 \circledast \mathbf{d}_2$. If the result would not be contained within AbsDOMData, the function is undefined.

Giving actions to the `item` and string manipulation commands requires the ability to calculate both the length of strings and length of forests. We give functions for this, which will also prove useful in defining logical expressions for DOM later (definition 81).

Definition 75 (String length)**.** Given the set of abstract DOM strings AbsDOMStrings (definition 73), the **string length** function $|\cdot|_s$: AbsDOMStrings $\rightharpoonup \mathbb{N}$ is defined as:

$$
\begin{array}{rl}
|\varnothing_s|_s \triangleq & 0 \\
|c|_s \triangleq & 1 \\
|\mathbf{s}_1 \cdot \mathbf{s}_2|_s \triangleq & |\mathbf{s}_1| + |\mathbf{s}_2| \\
\text{otherwise} & \text{undefined}
\end{array}
$$

Definition 76 (Forest length)**.** Given the set of abstract DOM forests AbsDOMForests (definition 73), the **forest length** function $|\cdot|_f : \text{AbsDOMForests} \rightharpoonup \mathbb{N}$ is defined analagously to definition 75.

With these, we can define the DOM commands, and give example actions. The majority of the complexity in the actions is in maintaining the structural invariants of the tree. For example, a document node must *only* be the root, and must *always* have exactly one child element node. The actions must preserve these invariants to avoid creating invalid data structure.

Definition 77 (DOM command actions)**.** Given a set of program variables PVars (definition 3), the atomic commands of DOM are those of the variable system (example 2), and the following additional commands: for all $\mathtt{s}, \mathtt{n}, \mathtt{m}, \mathtt{i}, \mathtt{c} \in \text{PVars}$, $ds \in \text{StructHeaps}$

$$
\begin{array}{rl}
\text{C} \triangleq & \mathtt{n := createElement(}e\mathtt{)} \mid \mathtt{n := createTextNode(}e\mathtt{)} \mid \mathtt{s := n.nodeName} \\
& \mid \mathtt{m := n.parentNode} \mid \mathtt{c := n.childNodes} \mid \mathtt{n := f.item(}e\mathtt{)} \\
& \mid \mathtt{o := n.removeChild(m)} \mid \mathtt{o := n.appendChild(m)} \\
& \mid \mathtt{s := n.substringData(}e\mathtt{,}\ e\mathtt{)} \mid \mathtt{n.appendData(}e\mathtt{)} \mid \mathtt{n.deleteData(}e\mathtt{,}\ e\mathtt{)}
\end{array}
$$

The actions of the variable commands are given in definition 16. The actions of the DOM commands `createElement`, `appendChild`, `childNodes` and `item` are defined as: for all $doc \in \text{DOMDocs}$, $g_1, g_2 \in \text{DOMGroves}$, $s, s_1, s_2 \in \text{DOMStrings}$, $f, f_1, f_2 \in \text{DOMForests}$, $\mathbf{t}_1, \mathbf{t}_2 \in \text{AbsDOMTrees}$, $\mathbf{d} \in \text{AbsDOMData}$, $fs, fs_1, fs_2 \in \mathcal{P}(\text{ForestIDs})$

$$[\![\texttt{n := createElement}(e)]\!](ds) \triangleq$$
$$\begin{cases} \{ds[\texttt{n} \mapsto n, \mathrm{D} \mapsto doc \;\&\; g_2] \mid n \in \textsc{NodeIDs}\} & \text{if} \quad \begin{array}{l} (\!|e|\!)(ds) = s, \# \notin s, \\ ds(\mathrm{D}) = doc \;\&\; g_1, \\ g_2 = g_1 \oplus s_n[\varnothing_f]_\varnothing \end{array} \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{o := n.childNodes}]\!](ds) \triangleq$$
$$\begin{cases} \{ds[\texttt{o} \mapsto o, \mathrm{D} \mapsto \mathbf{d}_1 \circledast t_1]\} & \text{if} \quad \begin{array}{l} ds(\mathrm{D}) = \mathbf{d}_1 \circledast t_1, t_1 = s_{ds(\texttt{n})}[f]_{fs_1}, \\ t_2 = s_{ds(\texttt{n})}[f]_{fs_2}, fs_1 \subseteq fs_2, o \in fs_2 \end{array} \\ \{ds[\texttt{o} \mapsto o, \mathrm{D} \mapsto doc_2]\} & \text{if} \quad \begin{array}{l} ds(\mathrm{D}) = doc_1, doc_1 = \#\text{document}_{ds(\texttt{n})}[t]_{fs_1} \;\&\; g, \\ doc_2 = \#\text{document}_{ds(\texttt{n})}[t]_{fs_2} \;\&\; g, \\ fs_1 \subseteq fs_2, o \in fs_2 \end{array} \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{o := f.item}(e)]\!](ds) \triangleq$$
$$\begin{cases} \{ds[\texttt{o} \mapsto o]\} & \text{if} \quad \begin{array}{l} ds(\mathrm{D}) = \mathbf{d} \circledast s_{1n}[f_1 \otimes s_{2o}[f_3]_{fs_2} \otimes f_2]_{fs_1}, ds(\texttt{f}) \in fs_1, \\ (\!|e|\!)(ds) = |f_1|_f \end{array} \\ \{ds[\texttt{o} \mapsto \textbf{null}]\} & \text{if} \quad \begin{array}{l} ds(\mathrm{D}) = \mathbf{d} \circledast s_n[f]_{fs}, ds(\texttt{f}) \in fs, \\ (\!|e|\!)(ds) < 0 \text{ or } (\!|e|\!)(ds) \geq |f|_f \end{array} \\ \{ds[\texttt{o} \mapsto o]\} & \text{if } ds(\mathrm{D}) = \#\text{document}_n[s_o[f]_{fs_2}]_{fs_1} \;\&\; g, ds(\texttt{f}) \in fs_1, (\!|e|\!)(ds) = 1 \\ \{ds[\texttt{o} \mapsto \textbf{null}]\} & \text{if } ds(\mathrm{D}) = \#\text{document}_n[t]_{fs} \;\&\; g, ds(\texttt{f}) \in fs, (\!|e|\!)(ds) \neq 0 \\ \{ds[\texttt{o} \mapsto \textbf{null}]\} & \text{if } ds(\mathrm{D}) = \mathbf{d} \circledast \#\text{text}_{ds(\texttt{f})}[s]_{fs}, ds(\texttt{l}) \in fs \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{o := n.appendChild(m)}]\!](ds) \triangleq$$
$$\begin{cases} \{ds[\texttt{o} \mapsto ds(\texttt{m}), \mathrm{D} \mapsto \mathbf{t}_1 \circledast \mathbf{t}_4 \textcircled{y} \varnothing_f]\} & \text{if} \quad \begin{array}{l} ds(\mathrm{D}) = \mathbf{t}_1 \circledast \mathbf{t}_2 \textcircled{y} t, \mathbf{t}_2 = s_{1\,ds(\texttt{n})}[\mathbf{f}]_{fs_1}, \\ t = s_{2\,ds(\texttt{m})}[f]_{fs_2}, \mathbf{t}_3 = s_{1\,ds(\texttt{n})}[\mathbf{f} \otimes t]_{fs_1} \end{array} \\ \{\lightning\} \text{ otherwise} \end{cases}$$

These cases span the types of actions needed for the DOM library. As with the tree library of section 3.1.2, the actions use the context structure to analyse the tree. The

`createElement` case simply extends the grove with a fresh node (the `createTextNode` case is similar). The `n.childNodes` case queries the DOM tree, and (due to the non-determinism), updates it. There are three cases. The first is when we can find the sub-tree identified by `n`, capturing it in t_1, and ignoring the surrounding context $_1$. The action updates the tree with some set of forest identifers that must include at least the original set fs_1, and returns an one of these selements. The second case performs the same task for the document node; this is needed to handle its non-uniform shape. The final case is when `n` does not reference a node within the tree. The rest of the query commands are in fact more simple than this case. The `item` command is typical of the index based access to lists of objects (the string manipulation commands are similar) .The `n.appendChild` command restructures the tree, and can fault based on the structural relationships of the parameters passed (the `n.removeChild` command is similar).

4.1.2. Examples

Here, we implement several addition DOM commands that are not present in the feather-weight subset. One example is `l := f.length`. This command returns the length of the forest identified by variable `n`.

```
l := f.length ≙ local r {
   l := 0;
   r := f.item(l);
   while (r != null)
    l := l + 1;
    r := f.item(l)
}
```

Another command is `v := n.value`. This command returns the *value* associated with a node. The only node type in our subset that with a value (in the DOM sense) is text nodes, whose values are the string stored within them. All other nodes return a **null** value. We implement this in two parts. The first obtains the length of a string, using a similar method to the length of a forest. The second uses the node name to discriminate between text nodes and other node types to ensure it can return the correct value.

```
l := n.stringLength ≜ local r {
 l := 0;
 r := n.substringData(l, 1);
 while (r != null)
  l := l + 1;
  r := n.substringData(l, 1)
}
```

```
s := n.value ≜ local l, name {
 name := n.nodeName;
 if name = "#text"
  l := n.stringLength;
  s := n.substringData(0, l)
 else
  s := null
}
```

Finally, we use this to create a `v := n.childValue` command. This command, not part of the DOM specification, obtains the first child of node `n`, and assigns the value of this child node to `v`. It will prove useful in the photo library example in section 4.4, allowing a more compact program.

```
v := n.childValue ≜ local cs, c {
   cs := n.childNodes;
   c := cs.item(0);
   v := c.value
  }
```

4.2. Reasoning about DOM

We now turn to reasoning about the DOM library. We present the reasoning as we did in the list and tree cases of section 3.1, first defining the structural addressing algebra for DOM, then giving the assertion language and axioms.

4.2.1. Abstract DOM data

The structural addressing algebra for DOM is built from the abstract DOM data (definition 73), compression function (definition 74), and the addresses function defined below.

Definition 78 (Addresses function for DOM)**.** Given the set of abstract DOM data AbsDOMData (definition 73) and the set of structural addresses StructAddrs, the **address function for DOM** addrs : AbsDOMData $\to \mathcal{P}$(StructAddrs) is

defined by induction to collect the set of structural addresses use within some abstract DOM data. It is similar to definition 37.

Lemma 15 (DOM contexts are structural addressing algebras)**.** *Given DOM data* DOMDATA *(definition 71), structural addresses* STRUCTADDRS *(definition 35), abstract DOM data* ABSDOMDATA *(definition 73), the DOM addresses function addrs (definition 78) and DOM compression comp (definition 74), the tuple:*

$$(\text{DOMDATA}, \text{STRUCTADDRS}, \text{ABSDOMDATA}, \mathit{addrs}, \mathit{comp})$$

is a structural addressing algebra.

Proof. Straightforward from the definitions. □

Abstract heaps for DOM are defined exactly as those for lists and trees were (sections 3.2.1 and 3.2.1 respectively). Notice that the DOM heap address D *still* maps to DOM documents, albeit abstract documents.

Definition 79 (Abstract heaps for DOM)**.** Given the structural addressing algebra for DOM (lemma 15), and program variables and values (definition 72), the set of **abstract heaps for DOM** are defined as:

$$\text{ABSHEAPS} \subseteq \begin{array}{c} (\text{PVARS} \xrightarrow{\text{fin}} \text{PVALS}) \sqcup (\{\text{D}\} \rightharpoonup \text{ABSDOMDOCS}) \\ \sqcup\ (\text{STRUCTADDRS} \xrightarrow{\text{fin}} \text{ABSDOMDATA}) \end{array}$$

4.2.2. Assertion language

We will reuse the abstract heap assertions of definition 55, and the standard framework assertions of definition 21 without change. The data assertion language for DOM is the lift of the data structure algebra of definition 73, plus the connectives required by the data assertions of structural separation logic (definition 57). We will, however, require a richer set of logical values and expressions to check various DOM invariants in the axioms.

Definition 80 (Logical values)**.** Given node identifiers NODEIDS, forest identifiers FORESTIDS and the null symbol **null** (parameter 18) and the set of abstract DOM data ABSDOMDATA (definition 73), the set of **logical values** LVARS is defined as:

$$\textsc{LVals} \triangleq \mathbb{B} \cup \mathbb{Z} \cup \textsc{NodeIDs} \cup \textsc{ForestIDs} \cup \{\textbf{null}\} \cup \mathcal{P}(\textsc{ForestIDs}) \cup \textsc{AbsDOMData}$$

The logical expressions consist of *length* expression, set operations, the *safe name* and *string* expressions, plus a compression expression. The length expression calculates the length of a DOM string or forest, and will be used to axiomatise `item` and the string commands. The standard set operations are used to analyse the sets of forest identifiers. The safeName(E) expression determines if E evaluates to a string with characters drawn DOMChars and not containing #. It will be used to check the validity of the name passed to `createElement`. The string(E) expression determines if the evaluation of E is a DOM string, and will be used to validate the parameter to `createTextNode`. The context application expression $\mathrm{E}_1 \circledcirc \mathrm{E}_2$ allows us to use compression and de-compression to to analyse the contents of logical variables, and will be used in the proofs of our examples.

Definition 81 (Logical expressions and evaluation)**.** Given a set of logical variables LVars (parameter 13) and logical values LVals (definition 130), the set of **logical expressions** LExprs, ranged over by $\mathrm{E}, \mathrm{E}_1, \cdots, \mathrm{E}_n$, is defined as the standard logical expressions (definition 14), and the following additional expressions:

$$\mathrm{E} ::= |\mathrm{E}]| \mid \mathrm{E}_1 \in \mathrm{E}_2 \mid \mathrm{E}_1 \cup \mathrm{E}_2 \mid \text{safeName}(\mathrm{E}) \mid \text{string}(\mathrm{E}) \mid \mathrm{E}_1 \circledcirc \mathrm{E}_2$$

The evaluation function $\langle\!|\cdot|\!\rangle(\Gamma)\cdot : \textsc{LEnvs} \to \textsc{LExprs} \rightharpoonup \textsc{LVals}$ is defined as that of the variable system (parameter 14), extended with:

$$
\begin{aligned}
(\!|\mathrm{E}|\!)(\Gamma) &\triangleq \begin{cases} |(\!|\mathrm{E}|\!)(\Gamma)|_s & \text{if } (\!|\mathrm{E}|\!)(\Gamma) \in \text{DOMStrings} \\ |(\!|\mathrm{E}|\!)(\Gamma)|_f & \text{if } (\!|\mathrm{E}|\!)(\Gamma) \in \text{DOMForests} \\ \text{undefined} & \text{otherwise} \end{cases} \\
(\!|\mathrm{E}_1 \in \mathrm{E}_2|\!)(\Gamma) &\triangleq \begin{cases} \textbf{true} & \text{if } (\!|\mathrm{E}_2|\!)(\Gamma) \text{ is a set}, (\!|\mathrm{E}_1|\!)(\Gamma) \in (\!|\mathrm{E}_2|\!)(\Gamma) \\ \textbf{false} & \text{if } (\!|\mathrm{E}_2|\!)(\Gamma) \text{ is a set}, (\!|\mathrm{E}_1|\!)(\Gamma) \notin (\!|\mathrm{E}_2|\!)(\Gamma) \\ \text{undefined} & \text{otherwise} \end{cases} \\
(\!|\mathrm{E}_1 \cup \mathrm{E}_2|\!)(\Gamma) &\triangleq \begin{cases} (\!|\mathrm{E}_1|\!)(\Gamma) \cup (\!|\mathrm{E}_2|\!)(\Gamma) & \text{if } (\!|\mathrm{E}_1|\!)(\Gamma), (\!|\mathrm{E}_2|\!)(\Gamma) \text{ are sets} \\ \text{undefined} & \text{otherwise} \end{cases} \\
(\!|\text{safeName}(\mathrm{E})|\!)(\Gamma) &\triangleq \begin{cases} \textbf{true} & \text{if } \begin{array}{l} (\!|\mathrm{E}|\!)(\Gamma) = s, s \in \text{DOMStrings}, \\ \nexists s_1, s_2 \in \text{DOMStrings}.\ s = s_1 \cdot \# \cdot s_2 \end{array} \\ \textbf{false} & \text{otherwise} \end{cases} \\
(\!|\text{string}(\mathrm{E})|\!)(\Gamma) &\triangleq \begin{cases} \textbf{true} & \text{if } (\!|\mathrm{E}|\!)(\Gamma) = s, s \in \text{DOMStrings}, \\ \textbf{false} & \text{otherwise} \end{cases} \\
(\!|\mathrm{E}_1 \mathbin{@} \mathrm{E}_2|\!)(\Gamma) &\triangleq (\!|\mathrm{E}_1|\!)(\Gamma) \mathbin{\textcircled{x}} (\!|\mathrm{E}_2|\!)(\Gamma), \mathbf{x} = \Gamma(\alpha)
\end{aligned}
$$

Definition 82 (DOM data assertions)**.** Given the set of logical variables LVars (definition 13) and logical expressions LExprs (definition 81), the set of **DOM data assertions** DataAssts, ranged over by ϕ, ϕ_1, ϕ_n, is defined by induction as follows: for all $\mathrm{N}, \mathrm{D}, \mathrm{FS}, \alpha \in \text{LVars}, \mathrm{E}, \mathrm{S} \in \text{LExprs}$

$$
\begin{aligned}
\phi ::= \quad & c \mid \phi_1 \cdot \phi_2 \mid \varnothing_s \mid \mathrm{S}_{\mathrm{N}}[\phi]_{\mathrm{FS}} \mid \#\text{text}_{\mathrm{N}}[\phi]_{\mathrm{FS}} \mid \phi_1 \otimes \phi_2 \mid \varnothing_f \mid \phi_1 \oplus \phi_2 \mid \varnothing_g \\
& \mid \#\text{document}_{\mathrm{D}}[\phi_1]_{\mathrm{FS}} \mathbin{\&} \phi_2 \mid \alpha \mid \phi_1 \mathbin{@} \phi_2 \mid \phi_1 \implies \phi_2 \mid \textbf{false} \mid \mathrm{E}
\end{aligned}
$$

We give the semantics of a DOM data assertion via a satisfaction relation, and will give the set interpretation of assertions in terms of this.

Comment 8. For large data assertion languages such as DOM, I have found this approach easier to understand than giving the set interpretation directly (as was done for the lists of section 3.2.1 and trees of section 3.2.1).

Definition 83 (Satisfaction relation for DOM data)**.** Given the set of logical environments LEnvs (definition 81), the abstract DOM data AbsDOMData (definition 73) and the DOM data assertions DataAssts (definition 82), the **DOM data assertion satisfaction relations** $\models$: (LEnvs $\times$ AbsDOMData) $\times$ DataAssts, with elements written $\Gamma, \mathbf{d} \models \phi$, is defined by induction as:

$$
\begin{array}{lcll}
\Gamma, \mathbf{s} \models & c & \iff & \mathbf{s} = c \\
\Gamma, \mathbf{s} \models & \phi_1 \cdot \phi_2 & \iff & \exists \mathbf{s}_1, \mathbf{s}_2.\ \mathbf{s} = \mathbf{s}_1 \cdot \mathbf{s}_2 \wedge \Gamma, \mathbf{s}_1 \models \phi_1 \wedge \Gamma, \mathbf{s}_2 \models \phi_2 \\
\Gamma, \mathbf{s} \models & \varnothing_s & \iff & \mathbf{s} = \varnothing_s \\
\Gamma, \mathbf{t} \models & S_{\text{N}}[\phi_2]_{\text{FS}} & \iff & \exists s, n, \mathbf{f}, fs.\ \begin{array}{l} s = (\!|S|\!)(\Gamma) \wedge n = \Gamma(\text{N}) \wedge fs = \Gamma(\text{FS}) \\ \wedge\ \mathbf{t} = s_n[\mathbf{f}]_{fs} \wedge \Gamma, \mathbf{f} \models \phi \end{array} \\
\Gamma, \mathbf{t} \models & \#\text{text}_{\text{N}}[\phi]_{\text{FS}} & \iff & \exists n, \mathbf{s}, fs.\ \begin{array}{l} n = \Gamma(\text{N}) \wedge fs = \Gamma(\text{FS}) \\ \wedge\ \mathbf{t} = \#\text{text}_n[\mathbf{s}]_{fs} \wedge \Gamma, \mathbf{s} \models \phi \end{array} \\
\Gamma, \mathbf{f} \models & \phi_1 \otimes \phi_2 & \iff & \exists \mathbf{f}_1, \mathbf{f}_2.\ \mathbf{f} = \mathbf{f}_1 \otimes \mathbf{f}_2 \wedge \Gamma, \mathbf{f}_1 \models \phi_1 \wedge \Gamma, \mathbf{f}_2 \models \phi_2 \\
\Gamma, \mathbf{f} \models & \varnothing_f & \iff & \mathbf{f} = \varnothing_f \\
\Gamma, \mathbf{g} \models & \phi_1 \oplus \phi_2 & \iff & \exists \mathbf{g}_1, \mathbf{g}_2.\ \mathbf{g} = \mathbf{g}_1 \oplus \mathbf{g}_2 \wedge \Gamma, \mathbf{g}_1 \models \phi_1 \wedge \Gamma, \mathbf{g}_2 \models \phi_2 \\
\Gamma, \mathbf{g} \models & \varnothing_g & \iff & \mathbf{g} = \varnothing_g \\
\Gamma, \mathbf{d} \models & \#\text{document}_{\text{D}}[\phi_1]_{\text{FS}}\ \&\ \phi_2 & \iff & \exists n, \mathbf{t}, fs, \mathbf{g}.\ \begin{array}{l} n = \Gamma(\text{D}), fs = \Gamma(\text{FS}), \\ \wedge\ \mathbf{d} = \#\text{document}_n[\mathbf{t}]_{fs}\ \&\ \mathbf{g} \\ \wedge\ \Gamma, \mathbf{d} \models \phi_1 \wedge \Gamma, \mathbf{g} \models \phi_2 \end{array} \\
\Gamma, \mathbf{d} \models & \alpha & \iff & \mathbf{d} = \Gamma(\alpha) \\
\Gamma, \mathbf{d} \models & \phi_1 \circledast \phi_2 & \iff & \exists \mathbf{d}_1, \mathbf{x}, \mathbf{d}_2.\ \begin{array}{l} \mathbf{x} = \Gamma(\alpha), \mathbf{d} = \mathbf{d}_1 \circledast \mathbf{d}_2 \wedge \\ \Gamma, \mathbf{d}_1 \models \phi_1 \wedge \Gamma, \mathbf{d}_2 \models \phi_2 \end{array} \\
\Gamma, \mathbf{d} \models & \phi_1 \Rightarrow \phi_2 & \iff & \Gamma, \mathbf{d} \models \phi_1 \Rightarrow \Gamma, \mathbf{d} \models \phi_2 \\
\Gamma, \mathbf{d} \models & \mathbf{false} & \iff & \text{never} \\
\Gamma, \mathbf{d} \models & \text{E} & \iff & \mathbf{d} = (\!|\text{E}|\!)(\Gamma)
\end{array}
$$

The standard first-order logic assertions are derived from $\Rightarrow$ and **false** in the standard way.

We define the following useful assertions.

Definition 84 (Derived assertions)**.** Given the DOM data assertions DOMAssts (definition 82) and the abstract heap assertions Assts (definition 55), we define the following **derived assertions for DOM**:

$$\begin{array}{rcl} \mathrm{S_N}[\phi] & \triangleq & \exists \mathrm{L}.\ \mathrm{S_N}[\phi]_{\mathrm{L}} \\ \mathrm{S}[\phi] & \triangleq & \exists \mathrm{N}, \mathrm{L}.\ \mathrm{S_N}[\phi]_{\mathrm{L}} \\ \Diamond_f \phi & \triangleq & \mathbf{true} \otimes \phi \otimes \mathbf{true} \\ \Box_f \phi & \triangleq & \neg \Diamond_f (\neg \phi) \end{array}$$

The first two assertions simply quantify over node or forest identifiers, and are used to reduce verbosity. The assertion $\Diamond_f \phi$ is read "*somewhere in this forest,* ϕ *holds*". It describes a DOM forest in which there is *some* sub-forest satisfying ϕ. The assertion $\Box_f \phi$ is the dual, read "*everywhere in this forest,* ϕ *holds*". It describes a forest in which *every* sub-forest satisfies ϕ (as it is not possible to find a sub-forest for which it does not hold).

So far, we have only given syntactic assertions. We now define the assertion interpretation function.

Definition 85 (Data assertion interpretation)**.** Given the set of DOM data assertions DataAssts (definition 82) and the logical environments LEnvs (definition 20), the set interpretation of a DOM assertion $(\!|\cdot|\!)^{\cdot} : \mathrm{DataAssts} \to \mathrm{LEnvs} \to \mathcal{P}(\mathrm{AbsDOMData})$ is defined as:

$$(\!|\psi|\!)^{\Gamma} \triangleq \{\mathbf{d} \mid \Gamma, \mathbf{d} \models \psi\}$$

Lemma 16 (Data assertions are valid)**.** *The DOM data assertions, and the interpretation above satisfy the behaviour required of the data assertions of structural separation logic in definition 57.*

Proof. Follows by the definition of the satisfaction relation, when compared to the requirements of definition 85. □

4.2.3. Axioms

We give syntactic axioms for the featherweight DOM library in figure 4.5. The associated semantic axioms are found by taking the union of the syntactic assertion interpretations under all logical variables. Atomic soundness of the axioms is similar to tree axioms shown sound in lemma 11.

The axioms are mostly straightforward. We examine a few axioms typical; the others are similar.

1. The `createElement(s)` command adds a new element to the grove. It has pre-condition $(\mathtt{n} \to \text{N} * \alpha \mapsto \varnothing_g * E) \wedge e \Rightarrow \text{S} \wedge \text{safeName(S)}$. It uses an abstract cell α as the location in which to place the new cell. That α is *not* deep within the DOM tree, but is instead at the grove level, is ensured by the pre-condition $\alpha \mapsto \varnothing_g$. Such an α can only have been allocated from the grove. To satisfy the DOM specification, we must ensure the name chosen for the node is valid; this is assured with safeName). The post-condition $\exists \text{N,FS}.\ \mathtt{n} \to \text{N} * \alpha \mapsto \text{S}_\text{N}[\varnothing_f]_\text{FS} * P$ asserts that the node is created, using the standard existential quantification for the new node and forest identifers.
2. There are many cases of `n.parentNode`, the command returning the identifier for the parent of `n`. The first two simply require that we provide an abstract cell containing the node `n` along with its parent; no other data is required, so is not used in the axiom. The more interesting is grove case (the third axiom), using pre-condition $\mathtt{n} \to \text{N} * \mathtt{m} \to \text{-} * \alpha \mapsto \#\text{document}_\text{N}[\beta]_\text{FS}\ \&\ \text{S}_\text{N}[\delta]_\text{FS2} \oplus \gamma$. Elements in the grove must return **null** when queries for their parent. However, if we used the pre-condition $\alpha \mapsto \text{S}_\text{N}[\beta]_\text{FS}$ to capture node `n`, the axiom could be used to show that every node has no parent! Instead, we capture the fact that grove nodes are siblings of the document node to ensure there is no ambiguity.
3. The `appendChild` axiom is similar to the equivalent command considered for trees in definition 62. However, here, the axiom leaves the assertion $\varnothing_f \vee \varnothing_g$ in place of `m` once moved. This is because we do not know if the node `m` has come from a forest of grove position. By using the disjunct, the choice is left to the *frame*. Once β is placed larger environment, and abstract deallocation used, the disjunct will collapse to either a forest or grove empty case.

Our axioms are significantly smaller and simpler than those of featherweight DOM [35]. For example, the `appendChild` axiom given there is:

$$\{\varnothing_f \multimap (\mathbf{c} \circ \text{PNM}_\text{P}[\text{PF}]_\text{PFID})) \circ \text{CNM}_\text{C}[\text{CF}]_\text{CFID}\}$$
$$\texttt{appendChild(p, c)}$$
$$\{(\mathbf{c} \circ (\text{PNM}_\text{P}[\text{PF}]_\text{PFID} \otimes \text{CNM}_\text{C}[\text{CF}]_\text{CFID})\}$$

In this axiom, a *covering context* $\mathbf{c}$ is used to capture the smallest tree with both `p` and `c` as descends. The need for this is discussed in section 1.2.3. Moreover, this axiom requires the adjoint of context composition to ensure that `c` is not a descendent of the

p. Our axiom does not require the covering context, reducing the footprint considerably. Similarly, the axiom for n.nodeName is given as:

$$\{\mathtt{s} \to \text{-} * \mathtt{n} \to \mathrm{N} * \mathrm{S_N}[\mathrm{F}]_{\mathrm{FS}}\}$$
$$\mathtt{s := n.nodeName}$$
$$\{\mathtt{s} \to \mathrm{S} * \mathtt{n} \to \mathrm{N} * \mathrm{S_N}[\mathrm{F}]_{\mathrm{FS}}\}$$

With structural separation logic, we do not need to include the entire sub-tree under the node being analysed. This is a clearer specification of the data accessed by the command.

Lemma 17 (Soundness)**.** *The DOM small axioms are atomically sound.*

The soundness argument is similar to those previous used for heaps, lists and trees.

4.2.4. Proofs of examples

Consider the l := n.length command implementation in 4.1.2. We prove that it satisfies the triple:

$$\{\mathtt{f} \to \mathrm{F} * \mathtt{l} \to \text{-} * \alpha \mapsto \mathrm{S_N}[\mathrm{CS}]_{\mathrm{FS}} \wedge \mathtt{f} \in \mathrm{FS}\}$$
$$\mathtt{l := f.length}$$
$$\{\mathtt{f} \to \mathrm{F} * \mathtt{l} \to |\mathrm{CS}| * \alpha \mapsto \mathrm{S_N}[\mathrm{CS}]_{\mathrm{FS}}\}$$

The specification is what would be expected of the command, and the proof the code meets it is given in figure 4.5. When given an element with a child forest identified by f, the result of the command is to return the length of the child forest. We follow the idioms introduced in chapter 3, mostly using the frame rule and semantic consequence (and hence abstract allocation) implicitly. This improves the readability of the proofs by eliding uninteresting steps, as the command footprints are generally evident simply by observation. However, in the this proof, we explicitly use semantic consequence and frame as a demonstration of the complete reasoning.

We pick a complex case, requiring the re-expression of resource using consequence before the sub-data is obviously available. Notice that the reasoning still follows the three idiomatic steps given in section 3.3.6: express the data in an equivalent de-compressable form; apply abstract allocation; and finally apply the axiom. Here, the chief work is in transforming the data representation so abstract allocation can be applied. This work would be equally necessary, and equally verbose, were normal separation or context logic being used, as the complexity is in managing the disjunction and logical variables.

The specification for the text node string length command states that the command, given a text node identifier, will return the length of the string contained within it. Note

$$\{(\mathtt{n} \to \mathrm{N} * \alpha \mapsto \varnothing_g * E) \wedge e \Rightarrow \mathrm{S} \wedge \mathrm{safeName}(\mathrm{S})\}$$
$$\mathtt{n := createElement}(e)$$
$$\{\exists \mathrm{N,FS}.\ \mathtt{n} \to \mathrm{N} * \alpha \mapsto \mathrm{S_N}[\varnothing_f]_{\mathrm{FS}} * E\}$$

$$\{(\mathtt{s} \to \text{-} * \mathtt{n} \to \text{-} * \alpha \mapsto \varnothing_g * E) \wedge e \Rightarrow \mathrm{S} \wedge \mathrm{string}(\mathrm{S})\}$$
$$\mathtt{n := createTextNode}(e)$$
$$\{\exists \mathrm{N,FS}.\ \mathtt{s} \to \mathrm{S} * \mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S}]_{\mathrm{FS}}\}$$

$$\{\mathtt{s} \to \text{-} * \mathtt{n} \to \mathrm{N} * \alpha \mapsto \mathrm{S_N}[\beta]_{\mathrm{FS}}\}$$
$$\mathtt{s := n.nodeName}$$
$$\{\mathtt{s} \to \mathrm{S} * \mathtt{n} \to \mathrm{N} * \alpha \mapsto \mathrm{S_N}[\beta]_{\mathrm{FS}}\}$$

$$\{\mathtt{s} \to \text{-} * \mathtt{n} \to \mathrm{N} * \mathrm{D} \mapsto \#\mathrm{document_N}[\alpha]_{\mathrm{FS}}\ \&\ \beta\}$$
$$\mathtt{s := n.nodeName}$$
$$\{\mathtt{s} \to \#\mathrm{document} * \mathtt{n} \to \mathrm{N} * \mathrm{D} \mapsto \#\mathrm{document_N}[\alpha]_{\mathrm{FS}}\ \&\ \beta\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \text{-} * \alpha \mapsto \mathrm{S_P}[\beta \otimes \mathrm{S'_N}[\delta]_{\mathrm{FS}_2} \otimes \gamma]_{\mathrm{FS}_1}\}$$
$$\mathtt{m := n.parentNode}$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{P} * \alpha \mapsto \mathrm{S_P}[\beta \otimes \mathrm{S'_N}[\delta]_{\mathrm{FS}_2} \otimes \gamma]_{\mathrm{FS}_1}\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \text{-} * \mathrm{D} \mapsto \#\mathrm{document_P}[\mathrm{S_N}[\delta]_{\mathrm{FS}_2}]]_{\mathrm{FS}_1}\ \&\ \beta\}$$
$$\mathtt{m := n.parentNode}$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{P} * \mathrm{D} \mapsto \#\mathrm{document_P}[\mathrm{S_N}[\delta]_{\mathrm{FS}_2}]]_{\mathrm{FS}_1}\ \&\ \beta\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \text{-} * \alpha \mapsto \#\mathrm{document_N}[\beta]_{\mathrm{FS}}\ \&\ \mathrm{S_N}[\delta]_{\mathrm{FS}_2} \oplus \gamma\}$$
$$\mathtt{m := n.parentNode}$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathbf{null}\alpha \mapsto \#\mathrm{document_N}[\beta]_{\mathrm{FS}}\ \&\ \mathrm{S_N}[\delta]_{\mathrm{FS}_2} \oplus \gamma\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \text{-} * \alpha \mapsto \#\mathrm{document_N}[\beta]_{\mathrm{FS}}\ \&\ \gamma\}$$
$$\mathtt{m := n.parentNode}$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathbf{null} * \alpha \mapsto \#\mathrm{document_N}[\beta]_{\mathrm{FS}}\ \&\ \gamma\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{c} \to \text{-} * \alpha \mapsto \mathrm{S_N}[\beta]_{\mathrm{FS}_1}\}$$
$$\mathtt{c := n.childNodes}$$
$$\{\exists \mathrm{FS}_2.\ \mathtt{n} \to \mathrm{N} * \mathtt{c} \to \mathrm{C} * \alpha \mapsto \mathrm{S_N}[\beta]_{\mathrm{FS}_2} \wedge \mathrm{FS}_1 \subseteq \mathrm{FS}_2 \wedge \mathrm{C} \in \mathrm{FS}_2\}$$

Figure 4.5.: Axioms for DOM (continued on on the following page)

$$\left\{\begin{array}{c} \mathtt{f} \to \mathrm{FID} * \mathtt{n} \to \text{-} * \alpha \mapsto \mathrm{S_M}[\mathrm{FS} \otimes \mathrm{S_N}[\gamma]_{\mathrm{FS}_2} \otimes \beta]_{\mathrm{FS}_1} * E \wedge e \Rightarrow \mathrm{I} \\ \wedge\ \mathrm{FID} \in \mathrm{FS}_1 \wedge \mathrm{I} = |\mathrm{FS}| \end{array}\right\}$$
$$\mathtt{n}\ \mathtt{:=}\ \mathtt{f.item(i)}$$
$$\left\{\ \mathtt{f} \to \mathrm{FID} * \mathtt{n} \to \mathrm{N} * \alpha \mapsto \mathrm{S_M}[\mathrm{FS} \otimes \mathrm{S_N}[\gamma]_{\mathrm{FS}_2} \otimes \beta]_{\mathrm{FS}_1} * E\ \right\}$$

$$\{\mathtt{f} \to \mathrm{FID} * \mathtt{n} \to \text{-} * \alpha \mapsto \mathrm{S_M}[\mathrm{FS}]_{\mathrm{FS}} * E \wedge e \Rightarrow \mathrm{I} \wedge \mathrm{FID} \in \mathrm{FS} \wedge \mathrm{I} < 0 \vee \mathrm{I} \geq \mathrm{FS}\}$$
$$\mathtt{n}\ \mathtt{:=}\ \mathtt{f.item}(e)$$
$$\{\mathtt{f} \to \mathrm{FID} * \mathtt{n} \to \mathbf{null} * \alpha \mapsto \mathrm{S_M}[\mathrm{FS}]_{\mathrm{FS}}\}$$

$$\left\{\begin{array}{c} \mathtt{f} \to \mathrm{FID} * \mathtt{i} \to \mathrm{I} * \mathtt{n} \to \text{-} * \alpha \mapsto \#\mathrm{document_M}[\mathrm{S_N}[\gamma]_{\mathrm{FS}_2}]_{\mathrm{FS}_1}\ \&\ \beta \\ \wedge\ \mathrm{FID} \in \mathrm{FS}_1 \wedge \mathrm{I} = 0 \end{array}\right\}$$
$$\mathtt{n}\ \mathtt{:=}\ \mathtt{f.item(i)}$$
$$\left\{\ \mathtt{f} \to \mathrm{FID} * \mathtt{i} \to \mathrm{I} * \mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{document_M}[\mathrm{S_N}[\gamma]_{\mathrm{FS}_2}]_{\mathrm{FS}_1}\ \&\ \beta\ \right\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{M} * \mathtt{o} \to \text{-} * \alpha \mapsto \mathrm{S_N}[\gamma \otimes \mathrm{S'_M}[\zeta]_{\mathrm{FS}_2} \otimes \delta]_{\mathrm{FS}_1} * \beta \mapsto \varnothing_g\}$$
$$\mathtt{o}\ \mathtt{:=}\ \mathtt{n.removeChild(m)}$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{M} * \mathtt{o} \to \mathrm{M} * \alpha \mapsto \mathrm{S_N}[\gamma \otimes \delta]_{\mathrm{FS}_1} * \beta \mapsto \mathrm{S'_M}[\zeta]_{\mathrm{FS}_2}\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{M} * \mathtt{o} \to \mathrm{O} * \alpha \mapsto \mathrm{S_N}[\gamma]_{\mathrm{FS}_1} * \beta \mapsto \mathrm{S'_M}[\mathrm{T} \wedge \mathit{is\_complete}]_{\mathrm{FS}_2} \wedge \mathrm{S} \neq \#\mathrm{text}\}$$
$$\mathtt{o}\ \mathtt{:=}\ \mathtt{n.appendChild(m)}$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{M} * \mathtt{o} \to \mathrm{O} * \alpha \mapsto \mathrm{S_N}[\gamma \otimes \mathrm{S'_M}[\mathrm{T}]_{\mathrm{FS}_2}]_{\mathrm{FS}_1} * \beta \mapsto (\varnothing_f \vee \varnothing_g)\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{M} * \mathtt{o} \to \mathrm{O} * \alpha \mapsto \mathrm{S_N}[\gamma]_{\mathrm{FS}_1} * \beta \mapsto \#\mathrm{text_M}[\gamma]_{\mathrm{FS}_2} \wedge \mathrm{S} \neq \#\mathrm{text}\}$$
$$\mathtt{o}\ \mathtt{:=}\ \mathtt{n.appendChild(m)}$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{m} \to \mathrm{M} * \mathtt{o} \to \mathrm{O} * \alpha \mapsto \mathrm{S_N}[\gamma \otimes \#\mathrm{text_M}[\gamma]_{\mathrm{FS}_2}]_{\mathrm{FS}_1} * \beta \mapsto (\varnothing_f \vee \varnothing_g)\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S} \cdot \mathrm{S'} \cdot \beta]_{\mathrm{FS}} * E \wedge e_1 \Rightarrow |\mathrm{S}| \wedge e_2 \Rightarrow |\mathrm{S'}|\}$$
$$\mathtt{s}\ \mathtt{:=}\ \mathtt{n.substringData}(e_1,\ e_2)$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{s} \to \mathrm{S'} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S} \cdot \mathrm{S'} \cdot \beta]_{\mathrm{FS}}\}$$

$$\{\mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S} \cdot \mathrm{S'}]_{\mathrm{FS}} * E \wedge e_1 \Rightarrow |\mathrm{S}| \wedge e_2 \Rightarrow \mathrm{J} \wedge \wedge \mathrm{J} > |\mathrm{S'}|\}$$
$$\mathtt{s}\ \mathtt{:=}\ \mathtt{n.substringData}(e_1,\ e_2)$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{s} \to \mathrm{S'} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S} \cdot \mathrm{S'}]_{\mathrm{FS}}\}$$

$$\{\mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text_N}[\beta]_{\mathrm{FS}} * E \wedge e \Rightarrow \mathrm{S}\}$$
$$\mathtt{n.appendData}(e)$$
$$\{\mathtt{n} \to \mathrm{N} * \mathtt{s} \to \mathrm{S} * \alpha \mapsto \#\mathrm{text_N}[\beta \cdot \mathrm{S}]_{\mathrm{FS}}\}$$

$$\{\mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S} \cdot \mathrm{S'} \cdot \beta]_{\mathrm{FS}} * E \wedge e_1 \Rightarrow |\mathrm{S}| \wedge e_2 \Rightarrow |\mathrm{S'}|\}$$
$$\mathtt{n.deleteData}(e_1,\ e_2)$$
$$\{\mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S} \cdot \beta]_{\mathrm{FS}}\}$$

$$\{\mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S} \cdot \mathrm{S'}]_{\mathrm{FS}} * E \wedge e_1 \Rightarrow |\mathrm{S}| \wedge e_2 \Rightarrow \mathrm{J} \wedge \mathrm{J} > |\mathrm{S'}|\}$$
$$\mathtt{n.deleteData}(e_1,\ e_2)$$
$$\{\mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S}]_{\mathrm{FS}}\}$$

Axioms for DOM (continued from figure 4.5 on the preceding page)

$\{\ \texttt{f} \to \mathrm{F} * \texttt{l} \to \texttt{-} * \alpha \mapsto \mathrm{NM_N}[\mathrm{CS}]_{\mathrm{FS}} \wedge \mathrm{F} \in \mathrm{FS}\ \}$

`l := f.length` $\triangleq$ `local r {`

`l := 0;`

We demonstrate a detailed analysis of `r := f.item(l)`

$\{\ \texttt{r} \to \texttt{-} * \texttt{f} \to \mathrm{F} * \texttt{l} \to 0 * \alpha \mapsto \mathrm{NM_N}[\mathrm{CS}]_{\mathrm{FS}} \wedge \mathrm{F} \in \mathrm{FS}\ \}$

The logical variable CS must have captured a forest

We perform case analysis on the possible values of this

It is either empty, or a node plus additional forest

$$\left\{\begin{array}{c} \texttt{r} \to \texttt{-} * \texttt{f} \to \mathrm{F} * \texttt{l} \to 0 * \alpha \mapsto \mathrm{NM_N}[\mathrm{CS}]_{\mathrm{FS}} \wedge \mathrm{F} \in \mathrm{FS} \\ (\mathrm{CS} = \varnothing_f \vee \exists \mathrm{S},\mathrm{M},\mathrm{CS2},\mathrm{FS2},\mathrm{R}.\ \mathrm{CS} = \mathrm{S_M}[\mathrm{CS2}]_{\mathrm{FS2}} \otimes \mathrm{R}) \end{array}\right\}$$

We then apply the rule of disjunction to consider both cases.

Assume for this detailed section that $\mathrm{CS} \neq \varnothing_f$.

$$\left\{\begin{array}{c} \texttt{r} \to \texttt{-} * \texttt{f} \to \mathrm{F} * \texttt{l} \to 0 * \\ \alpha \mapsto \mathrm{NM_N}[\mathrm{CS} \wedge \exists \mathrm{S},\mathrm{M},\mathrm{CS2},\mathrm{FS2},\mathrm{R}.\ \mathrm{S_M}[\mathrm{CS2}]_{\mathrm{FS2}} \otimes \mathrm{R}]_{\mathrm{FS}} \wedge \mathrm{F} \in \mathrm{FS} \end{array}\right\}$$

We must frame off the R forest.

$$\left\{\begin{array}{c} \texttt{r} \to \texttt{-} * \texttt{f} \to \mathrm{F} * \texttt{l} \to 0 * \\ \alpha \mapsto \mathrm{NM_N}[\mathrm{CS} \wedge \exists \mathrm{S},\mathrm{M},\mathrm{CS2},\mathrm{FS2},\mathrm{R}.\ \mathrm{S_M}[\mathrm{CS2}]_{\mathrm{FS2}} \otimes \beta \textcircled{\beta} \mathrm{R}]_{\mathrm{FS}} \wedge \mathrm{F} \in \mathrm{FS} \end{array}\right\}$$

Apply existential elimination

$$\left\{\begin{array}{c} \texttt{r} \to \texttt{-} * \texttt{f} \to \mathrm{F} * \texttt{l} \to 0 * \\ \alpha \mapsto \mathrm{NM_N}[\mathrm{CS} \wedge \mathrm{S_M}[\mathrm{CS2}]_{\mathrm{FS2}} \otimes \beta \textcircled{\beta} \mathrm{R}]_{\mathrm{FS}} \wedge \mathrm{F} \in \mathrm{FS} \end{array}\right\}$$

Apply abstract allocation.

We must record the structure of CS to reestablish the data invariant.

$$\left\{\begin{array}{c} \texttt{r} \to \texttt{-} * \texttt{f} \to \mathrm{F} * \texttt{l} \to 0 * \\ \exists \beta.\ \alpha \mapsto \mathrm{NM_N}[\mathrm{S_M}[\mathrm{CS2}]_{\mathrm{FS2}} \otimes \beta]_{\mathrm{FS}} * \beta \mapsto \mathrm{R} \wedge \\ \mathrm{F} \in \mathrm{FS} \wedge \mathrm{CS} = \mathrm{S_M}[\mathrm{CS2}]_{\mathrm{FS2}} \otimes \beta \textcircled{\beta} \mathrm{R} \end{array}\right\}$$

We can now apply the frame rule and the axiom

`r := f.item(l);`

We undo the above steps, and state the loop invariant

$$\left\{\begin{array}{c} \exists \mathrm{L},\mathrm{R},\mathrm{CS1},\mathrm{CS2}.\ \texttt{f} \to \mathrm{F} * \texttt{l} \to \mathrm{L} * \texttt{r} \to \mathrm{R} * \\ \left(\begin{array}{c} (\mathrm{R} = 0 \wedge \alpha \mapsto \mathrm{NM_N}[\mathrm{CS} \wedge \mathrm{CS1}]_{\mathrm{FS}}) \\ \vee \\ (\mathrm{R} \neq 0 \wedge \alpha \mapsto \mathrm{NM_N}[\mathrm{CS} \wedge \mathrm{CS1} \otimes \mathrm{R} \otimes \mathrm{CS2}]_{\mathrm{FS}}) \wedge \mathrm{L} = |\mathrm{CS1}| \end{array}\right) \wedge \mathrm{F} \in \mathrm{FS} \end{array}\right\}$$

```
while (r != null)
 l := l + 1;
 r := f.item(l);
}
```

$\{\ \texttt{f} \to \mathrm{F} * \texttt{l} \to |\mathrm{CS}| * \alpha \mapsto \mathrm{NM_N}[\mathrm{CS}]_{\mathrm{FS}}\ \}$

Figure 4.5.: Proof of `l := f.length`.

the use of the *is_complete* prediate to ensure the entire string contents are present for analysis. We show the code meets this specification in section A.1.2.

$$\{\texttt{n} \to \text{N} * \texttt{l} \to \text{-} * \alpha \mapsto \#\text{text}_{\text{N}}[\text{S} \wedge \mathit{is\_complete}]_{\text{FS}}\}$$
$$\texttt{l := n.stringLength}$$
$$\{\exists \text{L}.\ \texttt{n} \to \text{N} * \texttt{l} \to \text{L} * \alpha \mapsto \#\text{text}_{\text{N}}[\text{S}]_{\text{FS}} \wedge \text{L} = |\text{S}|\}$$

The node value command has the specification below. We a specification for the element and text node case, and show the code meets it (in the text node case) in figure A.1.

$$\{\texttt{n} \to \text{N} * \texttt{s} \to \text{-} * \alpha \mapsto \text{NM}_{\text{ID}}[\text{CS}]_{\text{FS}}\}$$
$$\texttt{s := n.value}$$
$$\{\texttt{n} \to \text{N} * \alpha \mapsto \text{NM}_{\text{ID}}[\text{CS}]_{\text{FS}} * (\text{NM} = \#\text{text} \wedge \texttt{s} \to \textbf{null}) \vee (\text{NM} \neq \#\text{text} \wedge \texttt{s} \to \textbf{null})\}$$

The first child value command has the following specification. The command is intended to extract the text from the very common XML idiom `<nodeName>Text value</nodeName>`. Notice that the pre-condition and post-condition clearly mirror this. The only complexity is in the update to the forest identifier set. We show the code meets this specification in the appendix, section A.1.4.

$$\{\texttt{n} \to \text{N} * \texttt{v} \to \text{-} * \alpha \mapsto \text{NM}_{\text{ID}}[\#\text{text}_{\text{TID}}[\text{VAL}]_{\text{FS1}} \otimes \beta]_{\text{FS2}}\}$$
$$\texttt{l := n.childValue}$$
$$\{\exists \text{FS3}.\ \texttt{n} \to \text{N} * \texttt{v} \to \text{VAL} * \alpha \mapsto \text{NM}_{\text{ID}}[\#\text{text}_{\text{TID}}[\text{VAL}]_{\text{FS1}}]_{\text{FS3}} \wedge \text{FS2} \subseteq \text{FS3}\}$$

4.3. Unsoundness in previous work

The existing work on featherweight DOM [35], [36] and thesis of Smith [65], gives the specification of `n.childNodes` as[2]:

$$\{\texttt{n} \to \text{N} * \texttt{c} \to \text{-} * \alpha \mapsto \text{S}_{\text{N}}[\beta]_{\text{F}}\}$$
$$\texttt{c := n.childNodes}$$
$$\{\texttt{n} \to \text{N} * \texttt{c} \to \text{F} * \alpha \mapsto \text{S}_{\text{N}}[\beta]_{\text{F}}\}$$

where F is a single forest identifier, rather than a set. As such, every call to `n.childNodes` returns the same forest identifier. This allows the derivation given in figure 4.6, where the

[2]Here updated to use our notation, but essentially identical.

specification states that the program always terminates in a state where variables `x` and `y` are identical. However, this behaviour is not guaranteed by the DOM specification.

That the specification returns the same identifier every time is too strong a statement. Whilst it is correct, in that any DOM implementation with this behaviour is correct, it is too restrictive, in that there are DOM implementations obeying the specification that do not behave in this way. We have seen this behaviour in the widely used WebKit browser engine[3]. This gives rise to a mismatch between the reasoning and DOM, as erroneous conclusions can be drawn about certain DOM programs[4]. The figure 4.6 shows a derivation which is false on some correct DOM implementations.

To support our interpretation, we examine the standard. In [71], the specification for the node interface contains a `childNodes` *attribute*:

```
interface Node {
  ⋮
 readonly attribute NodeList childNodes;
  ⋮
};
```

This attribute[5] is described as:

> *"A* `NodeList` *that contains all children of this node. If there are no children, this is a* `NodeList` *containing no nodes.*

Our implementation of `NodeList` is to return a forest identifer that references the node associated with the node list. This is acceptable, as we can see the forest identifier returned by `n.childNodes` as a simple "object" that just exposes the `item` command.

Even though declared as an attribute, the `childNodes` property of a node may return different results on each execution. To see this, we examine the specification for attributes given by Interface Description Language [54] (henceforth IDL). IDL is a language for describing object interfaces without committing an underlying implementation programming language, and is used by the DOM standard to define the node interfaces. On the behaviour of attributes, the IDL specification says:

[3] It was changed late in 2011 so that the NodeList objects resulting from calls are cached across requests (see Webkit bug 76591 [2]). The change was due to performance concerns, rather than standards compliance concerns.

[4] Although we have not yet found an instance of this problem in any examined code

[5] Note that `readonly` describes only that the attribute cannot be assigned to, not that it cannot change by other means.

> *"An interface can have attributes as well as operations; as such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions; one to retrieve the value of the attribute and one to set the value of the attribute."*

No restrictions are placed on the behaviour of the accessor functions. They are free to compute the result in any way that satisfies the documented behaviour of the object. As the `childNodes` property says only that the result is *"A NodeList that contains all children of this node"*, it is possible to comply with the specification by returning any object that gives the behaviour; the same one each time, a fresh one each time, or some combination of the two. Thus, the fragment:

```
if (n.childNodes != n.childNodes)
  print "An odd behavour"
```

will sometimes legitimately result in "An odd behaviour". The simple JavaScript program given in figure 4.6 can be used in any modern web browser to detect the issue.

In our model, we have corrected the problem by associating each node with a *set* of node list identifiers. The action of the `n.childNodes` command non-deterministically extends this set with zero or more fresh identifiers, and returns an element of the new set. This behaviour captures all possible outcomes: on each invocation, either a new identifier or an existing identifier is returned, with no way for the specification to prove which case occurs.

Comment 9. This is the only oversight I have discovered in the DOM specification given in [35], [65], and indicates only the innate complexity of interpreting specifications written in English, especially those dependent on other specifications. The error arose due to the unnatural behaviour of the `attribute` feature in IDL. Most people would have expected an "attribute" to be a consistent property of an object, such that `n.childNodes = n.childNodes` is always true. Whether this behaviour was intended by the IDL authors and not documented, assumed by the DOM authors and not validated, or is intentional, is unknown.

Two conclusions can be drawn here. The first is that specifications should not give words a different meaning to that which will be commonly understood by their audience. The second is that, ideally, authors of axiomatic specifications should validate as many of their assumptions as possible against implementations of the specification,

$\{\ \mathtt{n} \to \text{-} * \mathtt{cn1} \to \text{-} * \mathtt{cn2} \to \text{-} * \mathrm{D} \mapsto \#\mathrm{document}_{\mathrm{DN}}[\mathbf{true}]_{\mathrm{DFS}}\ \&\ \varnothing_g\ \}$

```
n := createElement("Test");
```

$\left\{\begin{array}{l} \exists \mathrm{N,FS}.\ \mathtt{n} \to \mathrm{N} * \mathtt{cn1} \to \text{-} * \mathtt{cn2} \to \text{-} * \\ \mathrm{D} \mapsto \#\mathrm{document}_{\mathrm{DN}}[\mathbf{true}]_{\mathrm{DFS}}\ \&\ \mathrm{Test}_{\mathrm{N}}[\varnothing_f]_{\mathrm{FS}} \end{array}\right\}$

```
cn1 := n.childNodes;
cn2 := n.childNodes;
```

$\left\{\begin{array}{l} \exists \mathrm{N,LFS}.\ \mathtt{n} \to \mathrm{DN} * \mathtt{cn1} \to \mathrm{FS} * \mathtt{cn2} \to \mathrm{FS} * \\ \mathrm{D} \mapsto \#\mathrm{document}_{\mathrm{DN}}[\mathbf{true}]_{\mathrm{DFS}}\ \&\ \mathrm{Test}_{\mathrm{N}}[\varnothing_f]_{\mathrm{FS}} \end{array}\right\}$

```
if (cn1 == cn2)
```

$\{\ \mathbf{true}\ \}$

```
  alert("The child node lists are the same");
else
```

$\{\ \mathbf{false}\ \}$

```
  alert("The child node lists differ");
```

$\{\ \mathbf{true}\ \}$

```
<!doctype html>
<html>
 <script>
  var n = document.createElement("Test");
  var cn1 = testEle.childNodes;
  var cn2 = testEle.childNodes;
  if (cn1 === cn2)
   alert("The child node lists are the same");
  else
   alert("The child node lists differ");
 </script>
</html>
```

Figure 4.6.: Demonstration of the specification unsoundness in the `n.childNodes` of previous DOM axiomatisations [35]. On WebKit versions prior to change-set 105372, this displays "The child node lists differ", in violation of the derivation given. Notation changes are due to the differences between our simple imperative language and JavaScript, and the fact our subset allows only one document.

and not just use the prose. When divergences are found, they can be investigated more deeply and either identified as a specification, implementation, axiomatisation bug.

4.4. Hybrid reasoning example

The DOM library is not used solely by web browsers. The ubiquity of libraries such as DOM has led to XML being adopted as a general *data transport format*. Many programs and protocols, rather than creating custom representations for data, now give output in XML. This allows other programs to consume their results using standard library implementations, rather than having to decode complex custom formats. XML, a text-only format, is even even used by programs storing non-text data such as photographs. Storing arbitrary data in XML requires an *encoding*, allowing it to be safely represented as text. One such encoding is *base-64*.

Base-64 is an encoding for data that would otherwise be stored as raw bytes (that is, base-256). Each of the 64 possible symbols in base-64 is represented with a *printable character*[6]. These are chosen to be a subset of most plain text formats, so base-64 allows arbitrary base-256 data to be stored in a text file. Each character in a text file typically uses 1 byte, so there is a size increase when transcoding from base-256 (8 useful bits per byte) to base-64 (7 useful bits per byte).

In this example, we consider a *photo library* XML file. This file represents the output of some image management tool. It stores a list of photographs, each associated with a name and the base-64 encoding of the photographic data. We will write and reason about a program that consumes this file, reading each photograph from the XML file, decoding the photo data from base-64 into a normal base-256 representation, and processing the decoded data. This program has several interesting facets as an example. It combines both DOM reasoning about the XML data with heap reasoning about the base-64 decoding of binary data. This demonstrates the hybrid reasoning of structural separation logic, something not straightforwardly achievable in the previous DOM work via context logic [35]. It is a typical use of the DOM library, analysing a list of nodes via assumptions about the document structure. The program also demonstrates the ability of structural separation logic to enable *parallel* DOM programs in a safe manner.

An example of the photo library XML manipulated by the program is given in figure

[6]Typically ‘A’ through ‘Z’ (upper-case letters), ‘a’ through ‘z’ (lower-case letters), ‘0’ through ‘9’ (numbers), and the characters ’+’ and ‘/’.

```
local f, i, p {
 i := 0;
 c := album.childNodes;
 p := c.item(i);
 while p ≠ null  local cs, nameNode, lenNode,
                       data64Node, name, len, data64, data {
  cs := p.childNodes;
  nameNode := cs.item(0);
  lenNode := cs.item(1);
  data64Node := cs.item(2);
  name := nameNode.childValue;
  len := lenNode.childValue;
  data64 := data64Node.childValue;
  // Allocate a heap buffer for the data, and decode into it
  data := alloc(len);
  base64Decode(data64, data);
  // Pass the name and buffer to the handler, and cleanup
  handleImage(name, data);
  freen(data, len);
  i := i + 1;
  p := c.item(i)
 }
}
```

Figure 4.7.: Example photo library decoder program.

4.8. The program proceeds by using the DOM library to iterate over the each `picture` child of the `album` element. For each picture, it will extract the name of the photo, the length of the photograph data in base-256, and the data's base-64 representation. It then uses the standard heap memory to allocate a buffer large enough to store the decoded representation of the photo data (as given by the `length` element). The base-64 data and the buffer pointer are passed to a `base64Decode` function that fills the buffer with the decoded photo data. The program then passes this, along with the name, to a handler function. The program text is given in figure 4.7.

Reasoning about this program requires handling both DOM data and normal flat heap cells. We thus extend the DOM heap in the natural way to account for flat heap cells:

$$\textsc{StructHeaps} \subseteq \begin{array}{c}(\textsc{PVars} \xrightarrow{\text{fin}} \textsc{PVals}) \sqcup (\mathbb{N}^+ \xrightarrow{\text{fin}} \mathbb{N}) \sqcup (\{\mathrm{D}\} \rightharpoonup \textsc{AbsDOMDocs}) \\ \sqcup\ (\textsc{StructAddrs} \xrightarrow{\text{fin}} \textsc{AbsDOMData})\end{array}$$

To represent an buffer, we use an inductive predicate *ncells*. The instance $\mathit{ncells}(\mathit{start}, \mathit{len}, \mathit{bytes})$ represents a contiguous sequence of heap memory cells, starting

at address *start*, continuing for *len* cells, and containing the list of bytes in *bytes*.

$$ncells(start, len, bytes) \triangleq \begin{array}{c} (len = 0 \wedge bytes = \epsilon \wedge \mathrm{emp}) \\ \vee \\ \left(\begin{array}{c} len > 0 \wedge \exists b, bytes'.\ bytes = b : bytes' \wedge \\ start \mapsto b * ncells(start + 1, len - 1, bytes') \end{array} \right) \end{array}$$

We create two additional heap axioms to represent allocating and freeing a new n-cell buffer.

$$\begin{array}{c} \{\texttt{c} \rightarrow \texttt{-} * \texttt{l} \rightarrow \mathrm{L}\} \\ \texttt{c := allocn(l)} \\ \{\exists \mathrm{C},\mathrm{D}.\ \texttt{c} \rightarrow \mathrm{C} * \texttt{l} \rightarrow \mathrm{L} * ncells(\mathrm{C}, \mathrm{L}, \mathrm{D})\} \end{array}$$

$$\begin{array}{c} \{\texttt{c} \rightarrow \mathrm{C} * \texttt{l} \rightarrow \mathrm{L} * ncells(\mathrm{C}, \mathrm{L}, \mathrm{D})\} \\ \texttt{freen(c, l)} \\ \{\texttt{c} \rightarrow \mathrm{C} * \texttt{l} \rightarrow \mathrm{L}\} \end{array}$$

For the purpose of this example, we assume a logical expression $decode_{64}(\mathrm{E})$ that describes the byte sequence obtained by base-64 decoding the evaluation of the logical expression E. We also assume an overload for the expression |E| that allows it calculate the length of a byte sequence. These are used to specify a command `base64Decode(data, buffer)` which takes a variable `data` containing base-64 encoded data, and a variable `buffer` pointing to a run of contiguous heap memory long enough to contain the base-256 representation of the `data` contents. The command decodes the data into the buffer, faulting if the buffer is not large enough.

$$\begin{array}{c} \{\texttt{data} \rightarrow \mathrm{D} * \texttt{buffer} \rightarrow \mathrm{B} * ncells(\mathrm{B}, \mathrm{LEN}, -) \wedge \mathrm{LEN} \geq |decode_{64}(\mathrm{D})|\} \\ \texttt{base64Decode(data, buffer)} \\ \{\texttt{data} \rightarrow \mathrm{D} * \texttt{buffer} \rightarrow \mathrm{B} * ncells(\mathrm{B}, \mathrm{LEN}, decode_{64}(\mathrm{D}))\} \end{array}$$

We now have sufficient predicates and commands to reason about the flat heap part of the example. We must still describe the contents of the album DOM tree, for which we use *schema* predicates. Schema predicates describe the general *shape* of a document, and are somewhat analogous to XML schema [40] or the XDuce assertions of [43]. We define the following predicates to describe the schema of an `album` document:

```
<album>
 <picture>
  <name>Landscape</name>
  <length>12333</name>
  <data>TG9ydW0gaXBzdW0uLg...</data>
 </picture>
 <picture>
  <name>Beach</name>
  <length>44782</name>
  <data>ZG9sb3Igc2l0IGFtZX...</data>
 </picture> ...
</album>
```

Figure 4.8.: Example photo album XML file containing base-64 encoded data. The `album` element has a list of `picture` elements for children. These pictures have three children: a name, the length of the photograph data in base-256, and the base-64 encoded photograph data.

$$
\begin{aligned}
sVal(id, name, val) &\triangleq \exists_{\mathrm{N,FS}}.\ name_{id}[\#\mathrm{text}_{\mathrm{N}}[val]_{\mathrm{FS}}] \\
pic\left(\begin{array}{l} id, name, \\ length, data \end{array}\right) &\triangleq \exists_{\mathrm{ID}_1,\mathrm{ID}_2,\mathrm{ID}_3}.\ \mathrm{picture}_{id}\left[\begin{array}{l} sVal(\mathrm{ID}_1, \mathrm{name}, name) \otimes \\ sVal(\mathrm{ID}_2, \mathrm{length}, length) \otimes \\ sVal(\mathrm{ID}_3, \mathrm{data}, data) \end{array}\right] \\
&\qquad \wedge\ length = |decode_{64}(data)| \\
existPic &\triangleq \varnothing \vee \exists^{\mathrm{ID,\,NAME,}}_{\mathrm{LEN,\,DATA.}}\ pic(\mathrm{ID}, \mathrm{NAME}, \mathrm{LEN}, \mathrm{DATA}) \\
album(n, f) &\triangleq \mathrm{album}_n[\Box_f existPic]_f
\end{aligned}
$$

The predicate $sVal(id, name, val)$ describes an element with identifier id and name $name$, and a single child text node containing the string val. It is used to describe the elements `name`, `length` and `data` in the album document. The predicate pic describes each `picture` element in the tree, and ensures the claimed data length is actually correct. Similarly, the $existPic$ describes either a picture with existentially quantified parameters, or the empty element. This is used in the $album(n, f)$ predicate, which describes an entire album element. It uses the "everywhere" assertion of definition 134 along with $existPic$ to assert that every child of the album element is either a picture element, or empty. This $album$ predicate can describe any instance of an album XML file with the shape given in figure 4.8.

After the decoding of the image data, the name and the resultant byte sequence are passed to a dummy command `handleImage` that represents some kind of processing. We

do not model what this command does, and assume it has the following specification, which indicates it does not alter the buffer:

$$\{\texttt{name} \to \text{S} * \texttt{buffer} \to \text{B} * \mathit{ncells}(\text{B}, \text{LEN}, \text{D})\}$$
$$\texttt{handleImage(name, buffer)}$$
$$\{\texttt{name} \to \text{S} * \texttt{buffer} \to \text{B} * \mathit{ncells}(\text{B}, \text{LEN}, \text{D})\}$$

The decoder program is given in figure 4.9. The pre-condition is $\texttt{album} \to \text{A} * \alpha \mapsto \mathit{album}(\text{A}, \text{F})$, and the post-condition is $\exists \text{F}.\ \texttt{album} \to \text{A} * \alpha \mapsto \mathit{album}(\text{A}, \text{F})$. This is an example of a specification saying very little, but the Hoare triple saying a lot. The pre- and post-condition that form the specification are equally a specification for `skip`. However, the value of the triple is in examining the code. This, along with the proof that it meets the given specification, (also in figure 4.9), states that the program does work (albeit work unobservable from the specification) and, whatever this is, it does it *safely*. Our memory safe interpretation of triples ensures that every DOM and heap command is used in both a memory safe and structurally correct manner.

There are two notable facets to the proof. The first facet is the use of abstract allocation and frame to isolate resource. We have mostly applied the rule implicitly, but leave in one explicit use labelled ① . This use allows the `while` loop, which intuitively focuses on exactly one `picture` element to actually focus the assertions on that element. In the proof, the use of semantic consequence to perform abstract allocation has allowed a concise restructuring of the heap; we need only one step to isolate and express the sub-data needed, and can then apply the frame rule as normal. The second facet is the mixture of heap and structured resource reasoning. The *ncell* inductive predicate exists alongside the abstract heap cells, allowing a homogeneous reasoning style for structured and standard heaps.

This type of proof is the strength of structural separation logic. We have mixed standard separation-logic-style reasoning using normal inductive predicates with reasoning about highly structured data presented entirely abstractly. Despite this, the style is very similar across the program. There are no jarring switches in reasoning style, the specifications remain readable, and the proof is as easy to manage as those of standard separation logic.

Towards concurrency

The DOM library is rarely used in concurrent settings. The standard provides no safety guarantees for concurrent usage, and web browsers do not currently support shared memory concurrency at all. However, the proof given above indicates that structural separation logic could help enable concurrent DOM. The photo library is an example where concur-

Figure 4.9.: Proof of photo library decoder program (continued on the following page).

$\{ \quad \exists \mathrm{D}.\ \texttt{name} \to \mathrm{NM} * \texttt{len} \to \mathrm{LN} * \texttt{data64} \to \mathrm{DT} * \texttt{data} \to \mathrm{D} * \mathit{ncells}(\mathrm{D}, \mathrm{LN}, \mathit{decode}_{64}(\mathrm{DT})) \quad \}$

```
// Pass the name and buffer to the handler, and cleanup
handleImage(name, data);
```

$\{ \quad \exists \mathrm{D}.\ \texttt{name} \to \mathrm{NM} * \texttt{len} \to \mathrm{LN} * \texttt{data64} \to \mathrm{DT} * \texttt{data} \to \mathrm{D} * \mathit{ncells}(\mathrm{D}, \mathrm{LN}, \mathit{decode}_{64}(\mathrm{DT})) \quad \}$

```
freen(data, len);
```

$\{ \quad \exists \mathrm{D}.\ \texttt{name} \to \mathrm{NM} * \texttt{len} \to \mathrm{LN} * \texttt{data64} \to \mathrm{DT} * \texttt{data} \to \mathrm{D} * \texttt{f} \to \mathrm{FID} * \texttt{i} \to \mathrm{I} \quad \}$

```
i := i + 1;
p := f.item(i)
```

Unframe, and apply deallocation

$$\left\{ \exists \mathrm{FID},\mathrm{FS},\mathrm{P},\mathrm{I}. \left(\begin{array}{l} \texttt{album} \to \mathrm{A} * \texttt{f} \to \mathrm{FID} * \texttt{i} \to \mathrm{I} * \texttt{p} \to \mathrm{P} * \texttt{cs} \to \text{-} * \texttt{nameNode} \to \text{-} * \\ \texttt{lenNode} \to \text{-} * \texttt{data64Node} \to \text{-} * \texttt{name} \to \text{-} * \texttt{len} \to \text{-} * \texttt{data64} \to \text{-} * \texttt{data} \to \\ \alpha \mapsto \left(\begin{array}{c} \mathit{album}(\mathrm{A}, \mathrm{FS}) \wedge \\ \mathit{album}_{\mathrm{A}}[(\mathrm{P} = \mathbf{null} \wedge \varnothing) \vee \\ (\exists \mathrm{PRE},\mathrm{PST},\mathrm{NM},\mathrm{LN},\mathrm{DT}.\ \mathrm{PRE} \otimes \mathit{pic}(\mathrm{P}, \mathrm{NM}, \mathrm{LN}, \mathrm{DT}) \otimes \mathrm{PST} \\ \wedge\ |\mathrm{PRE}| = \mathrm{I})]_{\mathrm{FS}} \end{array} \right) \end{array} \right) \right\}$$

```
  }
}
```

$\{ \quad \exists \mathrm{F}.\ \texttt{album} \to \mathrm{A} * \alpha \mapsto \mathit{album}(\mathrm{A}, \mathrm{F}) \quad \}$

Continued decoder example from figure 4.9 on the preceding page

rency could be helpful. Rather than processing each `picture` element in sequence, all picture elements could be processed in parallel. This could be achieved with only a mild change the program structure and proof; rather than using the `while` loop directly, replicate the loop body in parallel composition, one for each element. The safety of this is guaranteed by structural separation logic, as we can abstractly allocate each `picture` subtree into a separate cell, as is done in the proof. With context logic, such separation would not be possible. Along with Raad [59], we are currently examining the possibilities for concurrent DOM in detail.

4.5. Summary

This chapter has presented a detailed case study regarding using structural separation logic, replicating previous work on abstract structured data reasoning about DOM. We have demonstrated:

1. In this entire chapter, that structural separation logic can be used to provide an accurate specification of a real standard, and is thus at least as suited for this task as previous context logic work.
2. In section 4.2.3, that compared to previous work on context logic, the command axioms can be significantly smaller.
3. With the photo library (example 4.4), that *hybrid reasoning* mixing low- and high-level programming styles can be reasoned about. Both abstract allocation and nor-

mal inductive predicates can co-exist, along with different heap data types. This was not possible with previous techniques

4. With the photo library example and smaller axioms, that structural separation logic lays the foundations for concurrent reasoning.

5. Naturally stable promises

We have now demonstrated local reasoning for abstract data via structural separation logic. This is a useful step towards our goal of specifying libraries but, unfortunately, many libraries do not manipulate their data in an entirely local manner. Consider libraries with commands that use wide-ranging analysis to find data for update. These are distinct from the libraries we have so far seen, which access data via unique identifiers. For example, to remove an item from a tree, we identify the item via a unique integer. In DOM, each node is accessed via a unique node identifier, and each forest via a unique forest identifier.

In all these cases, the position in the structure at which an update occurs is entirely determined by searching the structure for the distinct "handle" to the data. In this chapter, we consider libraries which do not use unique identifiers. Commands will specify the data they update by analysing large parts of the entire structure. Even though this analysis seems global, such libraries can still benefit from local reasoning techniques, as the *updates* are still focused on sub-data. This chapter introduces techniques for *weakening* the locality provided by structural separation logic, allowing the reasoning to use some global information about data whilst still manipulating structures in a local fashion.

One common example is tree libraries which use *paths* to identify resources. Such situations are common, one ubiquitous example being the POSIX file systems library

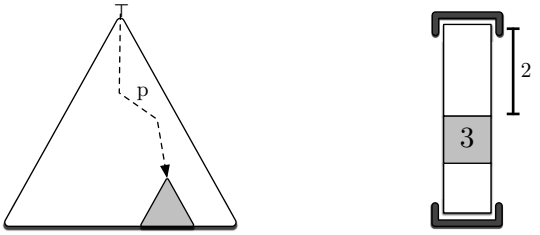


Figure 5.1.: On the left, a sub-tree found by path analysis in the larger tree. On the right, a sub-list found by index within the larger list.

(which we consider in chapter 6). Typically, paths analyse the tree structure from the root, descending until the end of the path, and so identify a sub-tree. Work is then performed on that sub-tree. These updates are, in some sense, *non-local*. The underlying resource needed for their safe execution is quite global, as the path analysis requires all the nodes along the path. However, this global resource is used *only* to identify a position in the structure where update will be performed. Another example is list elements that are accessed by index, similar to the `f.item(i)` command of DOM (chapter 4). Our axiom pre-condition for that command requires the entire list *up to* the element at index `i`. This prefix list is *global* information, not returned by the command, but needed to identify the local resource of interest. The commands use *global properties* to identify *local updates*. Figure 5.1 illustrates the intuitive footprints for two instances of these examples.

Structural separation logic's abstract allocation operation hides information about the context surrounding sub-data, making axiomatising these commands difficult. Given a sub-data, the only information we can deduce from it are the weak facts guaranteed by the structure of the model. For example, take the assertion $\alpha \mapsto 2$, which describes an element 2 within a list of distinct numbers. Because this data is a sub-list, the list model guarantees that it will be *completed* into a machine list. Ergo, we know that the frame must describe lists of the form $\mathrm{L} \mapsto [pl_1 \otimes \alpha \otimes pl_2]$ (albeit possibly split over many abstract heap cells). Moreover, as lists cannot contain duplicate elements, we know pl_1 and pl_2 cannot contain the element 2. This *implicit knowledge* is all we can deduce from the $\alpha \mapsto 2$ cell alone. It is not, for example, sufficient to determine that the element 2 is at some index i of the whole list.

This chapter considers extending the implicit knowledge about data with more *explicit knowledge* about the context into which it will fit. Consider figure 5.2 (a). The abstract heap cell $\mathbf{x}$ contains the partial tree $1[4 \otimes \mathbf{y}]$. Moreover, the data stored in the cell it is associated with two *promises*. The *head promise* above it indicates that whatever tree the data ultimately compresses into must place the data at path p. A *body promise* from the body address $\mathbf{y}$ states that, whatever sub-data ultimately compresses into $\mathbf{y}$, the path q exists within it. These promises allow the cell $\mathbf{x}$, when considered in isolation, to have strong guarantees about the context that will surround it. It cannot *mutate* the information in the promises, as they are read-only information provided by the frame. However, it can use the information in them, knowing that the set of possible frames has been restricted to those which match the shapes of the promises.

Now consider the $\mathbf{y}$ cell. We can see from the data within cell $\mathbf{x}$ that, whatever data is stored in cell $\mathbf{y}$, it must contain the path q. Mutations to the data at $\mathbf{y}$ must not change this fact, or the promise will have been broken. Therefore, when considered in isolation,

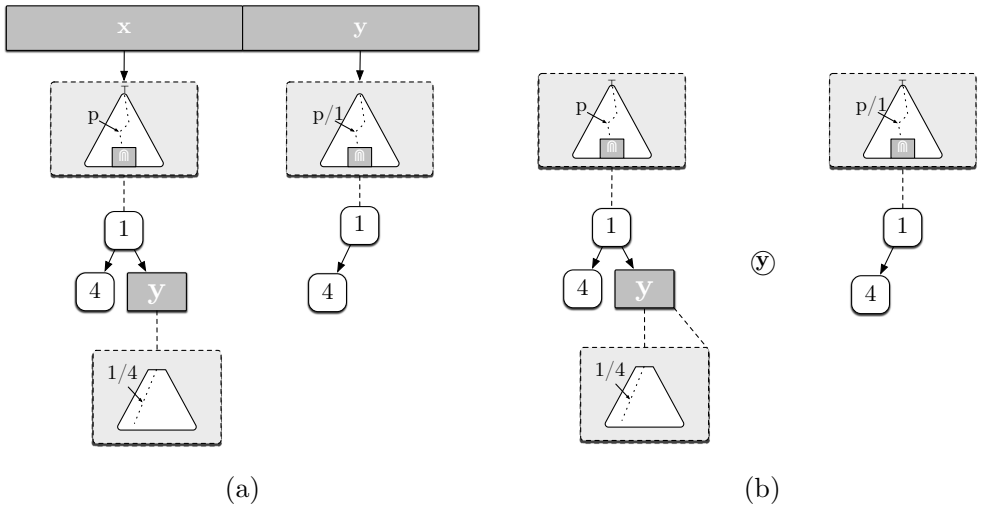


Figure 5.2.: In figure (a), we see an abstract heap holding two abstract cells with promises. The cell at **x** has the *head promise* that it will compress into data at path p, and a *body promise* from **y** that the sub-data compressing there will contain path $1/4$. The data at **y** is this sub-data, which has an *head promise* that it exists at path $p/1$. In figure (b), we see a single step of the collapse of this heap; this collapse checks that the sub-data satifies the promise $1/4$, and that the super-data satifies the promise $p/1$. As both hold, the compression is allowed.

the **y** cell must still enforce this restriction, or the system will be *unstable*. This chapter only considers *naturally stable* libraries, which have commands that can be axiomatised in a way that never breaks stability. In chapter 7, we will consider specifying libraries that are not naturally stable.

We give two examples of using promises. First, we demonstrate a tree library that identifies data with *linear paths* to show how promises can be used. Then, we adapt the list library of section 3.1.1 with an `item` command that allows list elements to be accessed by index.

Related work

Two strands of previous work are closely related to promises. The first is *Rely-Guarantee* reasoning [47], [69]. Rely guarantee allows the modelling of interference in concurrent reasoning through *relies* and *guarantees*. Hoare judgements take the form $R, G \vdash \{P\}\ \mathbb{C}\ \{Q\}$. Such a judgement states that $\{P\}\ \mathbb{C}\ \{Q\}$ holds if any concurrently executing *environment* program follows the relies R, and the program $\mathbb{C}$ follows the guarantees G. The relies and

guarantees are relations on the program state; R states the possible state updates the environment will do, G the program.

This work was adapted to *splittable* interference by *Deny-Guarantee* reasoning [26]. Whilst guarantees still state what a thread can do, *denies* state what the environment *cannot* do. Our promises act similarly to guarantees, as they state what the environment cannot do (that is, violate the promises). Careful choice of axioms ensures that deny perimssions are unneeded, as no guarantees can be violated. However, in chapter 7, we will see *obligations*, which are conceptually similar to deny permissions.

This chapter is thus perhaps more directly paralleled by *passivity systems*. Passive data, introduced to separation logic in [10], enables reasoning with *read-only* (or *passive*) data. The most commonly known formalism is fractional permissions. Ownership of a fractional heap cell $\mathtt{x} \xrightarrow{0.5} 5$ states that, whatever frame is added on, it cannot have more than half a fractional permission to the variable `x`. To ensure stability of the reasoning, no updates that could mutate `x` can be performed. A non-unity fraction acts both as a promise ("this heap cell will retain its value").

Promises extend this passivity to richer structures. They render structured data *passive with respect to sub-data shapes*. That is, certain mutations are allowed, as long as they preserve the shape information that has been shared via promises. Careful choice of axiomatisation ensures that when a promise has been issued, it cannot be violated. In this instance, they carry more information than just a binary "change/don't change" flag, due to the rich structure of the data.

5.1. Libraries with non-local data access

We begin by introducing two example libraries which will benefit from promises. The first, sibling unique trees, is a tree library where sub-trees are identified by paths from the root of the tree. The second, list indexing, extends our simple list library of section 3.1.1 with an *element access via index* command.

5.1.1. Sibling unique trees & paths

Our first library using global information to identify local data is *trees with paths*. In this library, sub-trees are identified not by unique handles, but by a *path* of nodes leading from the root of the tree to the sub-tree which will be analysed. We define sibling unique trees, the paths by which nodes in these trees can be identified, tree heaps, and some simple commands over trees.

Definition 86 (Trees with sibling unique identifiers)**.** The set SibTrees, ranged over by $st, st_1, \cdots, st_n$, is defined inductively as follows: for all $i \in \mathbb{N}^+$

$$st ::= i[st] \mid st_1 \otimes st_2 \mid \varnothing$$

where $\otimes$ is associative and commutative with identity $\varnothing$, and each tree contains sibling-unique tree identifiers. Trees are equal up to the properties of $\otimes$. We write nodes of the form $i[\varnothing]$ as i.

The set of **rooted sibling unique trees** SibRootedTrees, ranged over by $sr, sr_1, \cdots, sr_n$, is defined as the set of trees with exactly one root node:

$$\text{SibRootedTrees} = \{\top[st] \mid st \in \text{SibTrees}\}$$

Commands manipulating instances of this structure uniquely identify elements by a *path*. A path is a sequence of node identifiers.

Definition 87 (Tree path)**.** The set of **tree paths** TreePaths, ranged over by $pth, pth_1, \cdots, pth_n$, is defined as: for all $i \in \mathbb{N}^+$

$$pth ::= i \mid \top \mid i/pth \mid \top/pth$$

This definition allows the construction of paths like $1/\top/2/\top$. Whilst useless, these paths are harmless, as paths only identify resource by *resolution*. Resolution will never succeed when given such a path.

Definition 88 (Path resolution)**.** The **path resolution** function resolve : TreePaths $\to$ SibTrees $\rightharpoonup$ SibTrees is defined as:

$$\begin{aligned}
\text{resolve}(i, t_1 \otimes i[t_2]) &= i[t_2] \\
\text{resolve}(\top, \top[t]) &= \top[t] \\
\text{resolve}(i/pth, t_1 \otimes i[t_2]) &= \text{resolve}(pth, t_2) \\
\text{resolve}(\top/pth, \top[t]) &= \text{resolve}(pth, t) \\
\text{undefined} &\quad \text{in all other cases}
\end{aligned}$$

When $resolve(pth, t_1) = t_2$ is defined, we say that *pth resolves to* t_2 *in* t_1.

As an example, take the rooted tree $\top[2 \otimes 3[1[4 \otimes 5]]]$. In this tree, the path $\top/3/1$ resolves to the sub-tree $1[4 \otimes 5]$. The path $\top$ resolves to the entire tree. The path $\top/6$ does *not* resolve, as root node has no child 6.

The machines associated with these trees are identical to the tree machines of definition 30, except that they store rooted sibling unique trees.

Definition 89 (Rooted sibling-unique tree heaps)**.** Let $\mathrm{R} \notin \textsc{PVars}$ be the **tree heap address**. Then, the set of **rooted sibling-unique tree heaps** RootedSibTreeHeaps is defined as:

$$\textsc{RootedSibTreeHeaps} \triangleq (\textsc{PVars} \xrightarrow{\text{fin}} \textsc{PVals}) \sqcup (\{\mathrm{R}\} \to \textsc{SibRootedTrees})$$

The commands over tree heaps use paths to identify the resource they operate on. Two typical commands are below. We will see many more commands using paths in chapter 6.

1. `createNode(`e_p`,` e_n`)`: Creates a new node named with the evaluation of e_n, as a child of the node found by resolving the path e_p. If the path does not resolve, or resolves to a node already containing a child named e_n, the command faults.
2. `removeNode(`e_p`)`: Removes the tree found by resolving the evaluation of e_p from the tree. Faults if the evaluation of e_p does not resolve.

Each command identifies a location within the tree with a path from the root of the tree. To give axioms to these commands, we must demonstrate that the data we are working with is found at a given path. We will use axioms of the form:

$$\{e_p \Rightarrow \mathrm{P}/\mathrm{N} \land \alpha^{\mathrm{P}} \mapsto \mathrm{N}[\mathrm{T} \land \mathit{is\_complete}] * E\}$$
$$\texttt{removeNode(}e_p\texttt{)}$$
$$\{\alpha^{\mathrm{P}} \mapsto \varnothing * E\}$$

The pre-condition here describes that the path to the node to be removed is P/N, ergo the N child of the node found at path P will be removed. It also describes an abstract heap cell α, containing N, and associated with a *path promise* P. This promise ensures that α has been de-compressed from under the node at path P. Any frames that can be applied to this assertion *must not* contradict the fact that α is found at path P within the overall tree.

Such axioms split data into two parts: the normal *local data*, which will be used for update, and *global data* represented by promises, which is only used to place the local data in context. This global data will be shared with other heap cells, and may be *updated*

by other commands. For the promises associated with some data to make sense, we must ensure that they cannot be broken by updates. This is a *stability* property.

For the tree path promises of this example, this will be straightforward. Tree path promises are *naturally stable* with respect to the tree commands we give. Naturally stable promises are those which can never be invalidated by the commands of a library, thanks to a careful choice of axioms. With the two commands of this library, the only operation that could invalidate a path promise would be the removal of a sub-tree. We will axiomatise `removeNode` in a manner that ensures stability by checking that no abstract addresses are found beneath the node being removed. As promises are associated with structural addresses, if there are no body addresses in the sub-tree, no promises can be broken. Natural stability also applies to far more complex command sets. In chapter 6, we will show that a large subset of the POSIX file system library can be axiomatised in a naturally stable manner.

5.1.2. Lists access via indexing

We now extend the list library of section 3.1.1 with element access via *index*. We add a command much like the `j := f.item(`e`)` command of DOM:

`j := item(`e`)`: If e evaluates to an integer i, this command returns the element at index i within the list, faulting if either e does not evaluate to an integer, or the list has no item at index i.

Adding this command requires no changes to the underlying data or list machines. As with DOM, one natural choice of axiom is:

$$\{\alpha \mapsto [\mathrm{B} \otimes \mathrm{J} \otimes \beta] * \mathtt{j} \rightarrow \text{-} * E \wedge e \Rightarrow \mathrm{I} \wedge |\mathrm{B}| = \mathrm{I}\}$$
$$\mathtt{j := item}(e)$$
$$\{\alpha \mapsto [\mathrm{B} \otimes \mathrm{J} \otimes \beta] * \mathtt{j} \rightarrow \mathrm{J} * E\}$$

Considered in the light of promises, this axiom is unnecessarily large. The only fact needed by the command is that element J is at index I. The partial list B in the axiom is used only to demonstrate that fact. Consider instead a promise-bearing cell $\alpha^{\Subset \Leftarrow i} \mapsto \phi$. This describes an abstract heap cell α containing data associated with an *index promise* $\Subset \Leftarrow i$. The index promise states that the data has been allocated from some super-data list (indicated by symbol $\Subset$) at index i. With this promise, the command can be axiomatised as:

$$\left\{\alpha^{\Subset \Leftarrow \mathrm{I}} \mapsto \mathrm{J} * \mathtt{j} \rightarrow \text{-} * E \wedge e \Rightarrow \mathrm{I}\right\}\ \mathtt{j := item}(e)\ \left\{\alpha^{\Subset \Leftarrow \mathrm{I}} \mapsto \mathrm{J} * \mathtt{j} \rightarrow \mathrm{J} * E\right\}$$

This is a smaller axiom. It uses $\Uparrow \Leftarrow \mathrm{I}$ to describe that the node J has I left neighbours, but makes no statement about the contents of the neighbours. Critically, the axiom does not *own* those resources. It only "owns" the fact that they exist and that their length will not vary. The actual cells to the left remain free to be owned by another heap cell, as long as the number of them does not change. Again, the promise restricts the set of frames to ensure that α is only compressed into data that satisfies the promises.

Unfortunately, assertions using the promise $\Uparrow \Leftarrow \mathrm{I}$ are *not* stable when considered alongside the `remove` list command. Consider the assertion:

$$\mathrm{L} \mapsto [3 \otimes 1 \otimes 4 \otimes \alpha] * \alpha^{\Uparrow \Leftarrow 3} \mapsto 2$$

This describes valid heaps, α *is* at index 3 of list L. However, we can apply the frame rule to set aside $\alpha^{\Uparrow \Leftarrow 3} \mapsto 2$. This leaves no information that would prohibit us from removing, for example, element 3 from the list. Once the α cell is framed on again, it notices that its promise is broken. More concerning, it has been broken at *some point*, and when that was is unknown. The α cell may have used that promise whilst it was not actually true.

Comment 10. In general, it is actually worse than this thanks to the "*ABA*" problem commonly encountered in concurrent reasoning. For example, if the library had a command to add elements at arbitrary indexes, it may be that whilst α is allocated, the element 3 is removed, but then *restored* (it moves from state A to B to A again, hence the name of the problem). During the time it is in the B state, α still believes it is the third item of L, when it is not, but a check at compression time would still pass.

There is a strong relationship between the commands of a library, and the ability to create local axioms with naturally stable promises. Some pairings of commands, such a `remove` and `item`, have a tension between their ideal local axioms. In creating a local `item` specification, we have broken the local `remove` specification. We can restore correct behaviour by reducing the locality of either axiom: either `item` must be specified in terms of the entire list prefix that proves the index of an element, or `remove` must be specified in terms of the list prefix, to ensure it is not breaking the promises associated with an abstract heap cell.

In this chapter, we avoid this problem by restricting `remove` to only the *last* element of a list, essentially making it a random access stack. With this change, there are no commands that can break the index promises. To restore full locality to both commands,

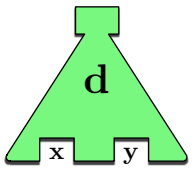


Figure 5.3.: A single datum $d \in \textsc{Data}$, for some $\textsc{Data}$ of a structural separation algebra.

chapter 7 adds *obligations* to the lists.

5.2. Promises

Promises are added to *data* to provide global *context* to each datum. They extend some choice of structural separation algebra, say $(\textsc{StructAddrs}, \textsc{Data}, \mathsf{addrs}, \mathsf{comp})$. Take some $d \in \textsc{Data}$, and assume it is *incomplete*, so that (for example) it can be composed into some super-data, and has two body addresses $\mathbf{x}$ and $\mathbf{y}$ into which sub-data can be composed. In figure 5.3, we see such a datum.

Such an incomplete datum can be *completed*. In structural separation logic, this happens as part of the completion process (described in section 3.4). To guide this process, we can augment data with the *shapes* of the data which will complete it. Call a super-data into which d collapses a *head* for d, and sub-data which collapse into $\mathbf{x}$ and $\mathbf{y}$ *bodies* for those addresses. Head and body data must *complete* d, so that each head data has exactly one hole, and each body data has no holes. We do not know, a-priori, the body address in the super-data into which d will collapse. Therefore, we will use a *placeholder* head address, $\Cap$. In figure 5.4, we see such a head datum and two body data.

This single head datum and pair of body data describes only one possibility for the context surrounding d. We describe the *shapes* of the contexts by gathering head and body data into sets. These sets represent the possible heads and bodies that extend d to be complete. So that we know which represents the head data, and which represent the body data, we *name* each set with the address to which it pertains. We use the head address $\Cap$ to name the head data, and the associated body address for each set of body data. These named sets are the *promises* associated with d, as seen in figure 5.5.

We call data paired with promises *promise-carrying data*. It is a new form of data, derived from some structural separation algebra and choice of promises. This new data will form a structural addressing algebra of its own (albeit with some minor differences, which we describe in section A.3). As such, it requires an addresses function and notion of

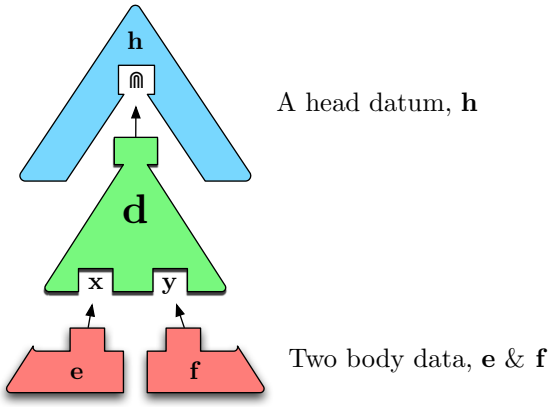


Figure 5.4.: The data d can be extended by a super-data and two sub-data to be *complete*. We call these types of data *head* and *body* data respectively.

compression. The addresses function is straightforward, simply considering the addresses of the data that is paired with the promises.

Compression is more interesting. It requires that we *check* the promises are true. Consider figure 5.6, which shows the compression between the promise-carrying data $(d, \{⋒ : p_1, \mathbf{x} : p_2, \mathbf{y} : p_3\})$ and sub-data $(f, \{⋒ : p_4\})$, using the address $\mathbf{y}$, where each p_i represents the set of data associated with each promise. The super-data d has a body promise $\mathbf{y} : p_3$ for the $\mathbf{y}$ address. The sub-data is carrying a head promise $⋒ : p_4$, describing the complete super-data that will fit over it. It has no body addresses, and so no body promises.

Compression must do three things:

1. Check the body promise for $\mathbf{y}$ associated with d, $\mathbf{y} : p_3$, is satisfied by the sub-data f.
2. Check that the head promise associated with f, $\mathbf{y} : p_3$, is satisfied by the super-data d.
3. Compress the underlying data and remove redundant promises.

Checking that the promises are satisfied involves comparing the data d and $f$with the appropriate promise. In this instance, checking that f satisfies the promises associated with the $\mathbf{y}$ hole in d is straightforward. Because f is already complete, we need only ensure that f is in p_3. Checking that d satisfies the head promise associated with f is harder, as d has more than just the $\mathbf{y}$ body address, and will still be extended by some

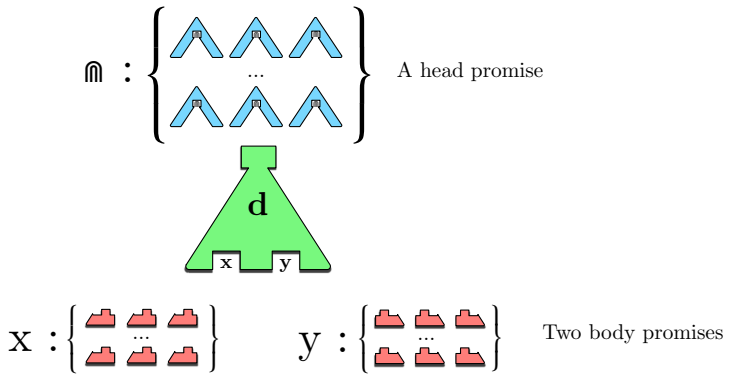


Figure 5.5.: By collecting head data and body data into sets and associating these sets with *names*, we can describe the shape of the context for d. These named sets are *promises*.

super-data. However, it can be *considered complete* by way of its promises. To check that d satisfies the head promise p_4, we use *closure with respect to* $\mathbf{y}$. Closure uses promises to *fill in* missing data. It is with respect to a specific body address, into which the address ⋔ will be placed. This ensures that the closures matches the shapes used for head promises. The completion here is illustrated in figure 5.7. The result of this closure can be used to check the promises p_4 hold by standard subset inclusion.

Had we been composing d into some super-data we would have used *body closure* to complete it, thus allowing the promises associated with the super-data to be checked. As normal closure is used to check head promises, body closure is used to check body

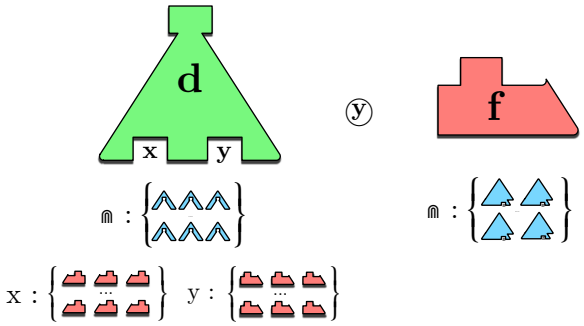


Figure 5.6.: The compression of promise-carrying data d with sub-data f using address $\mathbf{y}$.

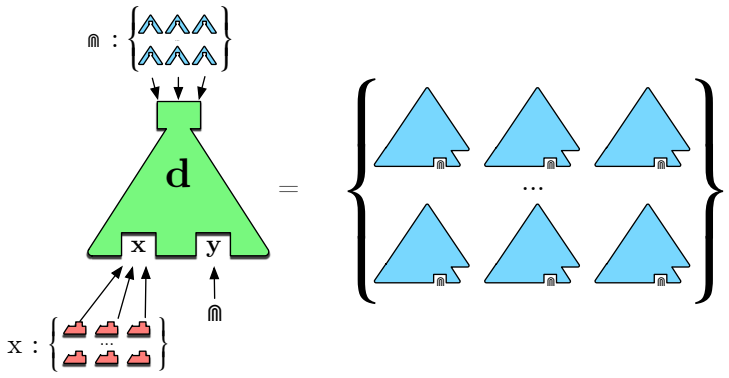


Figure 5.7.: The closure of a promise-carrying data d with respect to $\mathbf{y}$. Each head data in the head head promise, and body data int

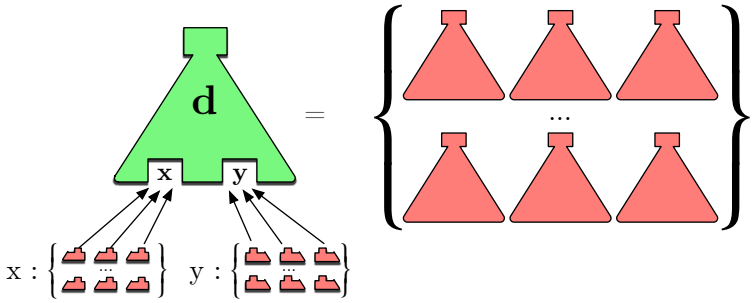
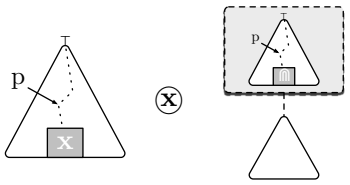


Figure 5.8.: The body closure of a promise-carrying data fills each body address using the promises associated with them.

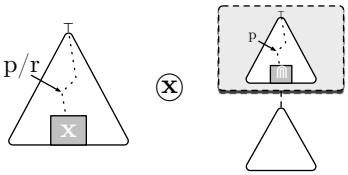
promises. These have no addresses, so this closure fills all addresses, an shown in figure 5.8.

If the promises pass the checks, so that f is within p_3, and p_4 is a subset of the closure of d with respect to $\mathbf{y}$, we can perform the compression. The underlying data is compressed using standard data compression. The promises are *merged*, with the head promise for f discarded (as it now has a head), and the body promise associated with $\mathbf{y}$ discarded (as the address no longer exists). The result is $(d \textcircled{\mathbf{y}} f, \{⋔ : p_1, \mathbf{x} : p_2\})$.



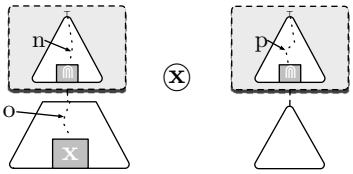
✓

Compression is allowed, as the super-data is a complete rooted tree, where the address **x** is at path p, which matches the head promise associated with the sub-data.



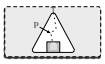
✗

Compression fails, as the super-data involved in compression does not have the address at path p, and so does not fulfil the promise.



?

The success of compression depends on the values of n and o. If $p = n/o$, the compression succeeds, as the super-data provides the o part of the promise, and uses its own promise to provide the n. If $p \neq n/o$, then compression fails.

Figure 5.9.: Examples of promises for linear paths. The promise  indicates that the data is at path p in the tree.

Path promises for sibling-unique trees

As an example of promise-carrying data, consider figure 5.9. This considers the *path promises* for the trees with paths library. It shows the combinations of promises that allow or disallow compression in a variety of cases.

5.2.1. Formalising promises

We now formalise the intuitions of promises given in the last section. Promises are added to a choice of structural addressing algebra. We will require a method for identifying the single body addressed used in head data. As this address is, in essence, a placeholder for the real address which will be used, we nominate a *placeholder address* from the data. We remove this from the structural addressing algebra to ensure it is not used in describing data. This creates two algebras: The original *full* algebra which will describe promises, and the *underlying* algebra, which will describe data (and cannot use the head address).

Definition 90 (Underlying data algebra)**.** Let $(\textsc{StructAddrs}, \textsc{Data}, \mathsf{addrs}, \mathsf{comp})$ be a structural separation algebra where $\Cap \in \textsc{StructAddrs}$. The derived **underly-**

ing data algebra is defined as:

$$(\textsc{StructAddrs}_{\Cap}, \textsc{Data}_{\Cap}, \text{addrs}_{\Cap}, \text{comp}_{\Cap})$$

where $\Cap$ is distinguished as the **head address**, and

1. $\textsc{StructAddrs}_{\Cap} = \textsc{StructAddrs} \setminus \{\Cap\}$
2. $\textsc{Data}_{\Cap} = \{d \in \textsc{Data} \mid \Cap \notin \text{addrs}(d)\}$
3. $\text{addrs}_{\Cap} = \text{addrs}$
4. $\text{comp}_{\Cap} = \text{comp}$

Notice that the underlying data algebra will remain a structural separation algebra, as we have only disabled one address out of a countable infinity. The arbitrary addresses property of the algebra will ensure that any use of it can be replaced by another of the addresses.

We now define a set of *head data*, which are all those sets of data with elements that have only a single body address $\Cap$. These head data will describe the *shape* of possible head promises. The *vacuous head data* which describes *every possible shape* of super-data that some data can compress into. It will be used to stand in for a missing promise, as it tells data no more than it already knew implicitly.

Definition 91 (Head data)**.** Given a structural addressing algebra $(\textsc{StructAddrs}, \textsc{Data}, \text{addrs}, \text{comp})$ and underlying data algebra using head address $\Cap$, the set of **head data** $\textsc{HeadData}$, ranged over by $\overline{d}, \overline{d}_1, \cdots, \overline{d}_n$, is defined as:

$$\textsc{HeadData} = \{\overline{d} \in \mathcal{P}(\textsc{Data}) \mid \forall d \in \overline{d}.\ \text{addrs}(d) = \{\Cap\}\}$$

The **vacuous head data** $\textsc{VacHeadData} \in \textsc{HeadData}$ is defined as:

$$\textsc{VacHeadData} = \{d \in \textsc{Data} \mid \text{addrs}(d) = \{\Cap\}\}$$

The set of *head promises* are the pairs of the head address and head data. Here, the head address acts as the *name* for the promise. With respect to some data d, a head promise $(\Cap, \overline{d})$ is read as *"The data d is promised that the head eventually fitting over it is contained within $\overline{d}$"*. Notice this does *not* say "All data within a head promise fit over the data". Promises are conservative approximations of the surrounding context of data.

Parameter 19 (Head promises)**.** Given the set of head data HeadData (definition 91), assume a set of **head promises** HeadProms, ranged over by $\pi^H, \cdots$, with the type:

$$\textsc{HeadProms} \subseteq \{\Cap\} \times \textsc{HeadData}$$

We now turn to body promises. Body promises are similar to head promises, but describe the shape of the *sub-data* that fit within a datum. These shapes are again described by sets of data, this time called *body data*. To ensure that body promises result in complete data, body data they must contain no addresses. We again describe a vacuous body data, used to stand in for missing body promises.

Definition 92 (Body data)**.** Given a structural addressing algebra $(\textsc{StructAddrs}, \textsc{Data}, \text{addrs}, \text{comp})$ and underlying data algebra using head address $\Cap$, the set of **body data** BodyData, ranged over by $\underline{d}, \underline{d}_1, \cdots, \underline{d}_n$, is defined as:

$$\textsc{BodyData} = \{\underline{d} \in \mathcal{P}(\textsc{Data}) \mid \forall d \in \underline{d}.\ \text{addrs}(d) = \emptyset\}$$

The **vacuous body data** VacBodyData $\in$ BodyData is defined as:

$$\textsc{VacBodyData} = \{d \in \textsc{Data} \mid \text{addrs}(d) = \emptyset\}$$

Body promises are pairs of structural addresses and body data. With respect to some data d, a body promise $(\mathbf{x}, \underline{d})$ is read "The body that fits into address $\mathbf{x}$ in d is contained within $\underline{d}$".

Parameter 20 (Body promises)**.** Given the set of body data BodyData (definition 92), assume a set of body promises BodyProms, ranged over by π^B, with the type:

$$\textsc{BodyProms} \subseteq \textsc{StructAddrs} \times \textsc{BodyData}$$

Given data d and a set of body promises, we can perform *body promise closure* of the data. This compresses data into every address within d. If there is a body promise associated with that address, it uses the promised data. If there is no body promise for that address, it uses the vacuous body data. Body promise closure is used during

promise-carrying data compression to check that promises are met.

Definition 93 (Body promise closure of data)**.** Given a structural addressing algebra (StructAddrs, Data, addrs, comp), associated underlying data algebra using head address $\Cap$, and body promises BodyProms (parameter 20), the **body promise closure** function $\downarrow\!\!\!\downarrow$: $\text{Data}_{\Cap} \to \mathcal{P}(\text{BodyProms}) \to \mathcal{P}(\text{Data})$ is defined as:

$$\downarrow\!\!\!\downarrow (d, \Pi^B) = \left\{ d \textcircled{\scriptstyle \mathbf{x}_1} d_1 \textcircled{\scriptstyle \mathbf{x}_2} \cdots \textcircled{\scriptstyle \mathbf{x}_n} d_n \;\middle|\; \begin{array}{l} \text{addrs}(d) = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}, i \in \{1, \cdots, n\} \\ \exists \text{unique } \underline{d}_i.\ (\mathbf{x}_i, \underline{d}_i) \in \Pi^B \implies d_i \in \underline{d}_i, \\ \nexists \underline{d}_i.\ (\mathbf{x}_i, \underline{d}_i) \in \Pi^B \implies d_i \in \text{VacBodyData} \end{array} \right\}$$

This function is well-defined by the quasi-commutativity of compression on the underlying data.

By combining head and body promises, we obtain a general set of *promises*. From this, we draw *promise sets*, which are subsets containing head and body promises, where each promise name is used at most once. When clear, we will call these promise sets call simply "promises".

Definition 94 (Promises)**.** Let HeadData $\cup$ BodyData be ranged over by $\overline{\underline{d}}, \overline{\underline{d}}_1, \cdots, \overline{\underline{d}}_n$. Then, set of **promise sets**, ranged over by $\Pi, \Pi_1, \cdots, \Pi_n$, is defined as:

$$\text{Promises} = \left\{ \Pi \in \mathcal{P} \left(\begin{array}{c} \text{HeadProms} \\ \cup \\ \text{BodyProms} \end{array} \right) \;\middle|\; \begin{array}{l} \nexists \mathbf{x} \in \text{StructAddrs}. \\ \exists \overline{\underline{d}}_1, \overline{\underline{d}}_2.\ \{(\mathbf{x}, \overline{\underline{d}}_1), (\mathbf{x}, \overline{\underline{d}}_2)\} \subseteq \Pi \end{array} \right\}$$

Let $\Pi \in$ Promises. We write the removal of the promise from any address $\mathbf{x} \in$ StructAddrs from Π as:

$$\Pi - \mathbf{x} \triangleq \{(\mathbf{y}, \overline{\underline{d}}) \mid (\mathbf{y}, \overline{\underline{d}}) \in \Pi, \mathbf{y} \neq \mathbf{x}\}$$

Promise sets are associated with data drawn from the underlying data algebra, and provide context for each datum. We call the pairing of data and a set of promises *promise-carrying data*. Promises are optional, so that the promises paired with a datum need not have a promise associated with the head address, or any given body address. However,

every promise must be useful, so must be either a head promise, or a body promise named for a address within the datum.

Definition 95 (Promise-carrying data)**.** The set of **promise-carrying data** PromData, ranged over by $\mathbf{pd}, \mathbf{pd}_1, \cdots, \mathbf{pd}_n$ is defined as:

$$\textsc{PromData} \subseteq \textsc{Data}_{\Cap} \times \textsc{Promises}$$

where every promise is either a head promise, or a promise from some address in the data: for all $(d, \Pi) \in \textsc{PromData}$

$$\forall (\mathbf{x}, \overline{\underline{d}}) \in \Pi.\ \mathbf{x} = \Cap \text{ or } \mathbf{x} \in \text{addrs}(d)$$

When defining compression, we will need to check if data satisfies promises. When checking body promises, we will use the body closure of definition 93. This body closure uses promises to fill in body addresses. It produces results with the same type as body promises: sets of complete data. When checking head promises, we will use *general closure with respect to an address*. General closure uses promises to complete data, resulting in sets matching the type of head data: complete heads, with a single body address $\Cap$. We achieve general closure with respect to $\mathbf{x}$ by first using body closure on the body, whilst ensuring it fills the address $\mathbf{x}$ with $\Cap$. We then fit each head promise over the result.

Definition 96 (General closure of promise-carrying data)**.** Given a promise-carrying datum $\mathbf{pd} \in \textsc{PromData}$, the **closure of pd with respect to x** $\circlearrowleft_{\mathbf{x}}(\cdot) : \textsc{StructAddrs} \to \textsc{PromData} \to \mathcal{P}(\textsc{Data})$ is defined as:

$$\circlearrowleft_{\mathbf{x}}((d, \Pi)) = \bigcup \left\{ \overline{d} \Cap \underline{d} \;\middle|\; \begin{array}{l} \overline{d} \in \pi^H, \\ \underline{d} \in\!\downarrow (d, (\Pi - \Cap - \mathbf{x}) \cup \{(\mathbf{x}, \{\Cap\})\}) \end{array} \right\}$$

where $\pi^H = \pi^{H'}$ if $\exists \pi^{H'}.\ (\Cap, \pi^{H'}) \in \Pi$, and $\pi^H = \textsc{VacHeadData}$ otherwise.

This function is well defined via the quasi-commutativity of the underlying data, and the address irrelevance property of the data.

We can now define compression between promise-carrying data, $\mathbf{pd}_1 \circledast \mathbf{pd}_2$. Compression not only compresses the underlying data, but *checks promises*.

Definition 97 (Compression for promise-carrying data)**.** The **compression function** for promise-carrying data, comp_P : StructAddrs $\to$ PromData $\to$ PromData $\rightharpoonup$ PromData is defined as: for all $\mathbf{pd}_1 = (d_1, \Pi_1)$ and $\mathbf{pd}_2 = (d_2, \Pi_2)$

$$\text{comp}_P(\mathbf{x}, \mathbf{pd}_1, \mathbf{pd}_2) = (\text{comp}_{\Cap}(\mathbf{x}, d_1, d_2), (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap))$$

where:

1. The underlying data compression is defined: $\text{comp}_{\Cap}(\mathbf{x}, d_1, d_2)$ defined.
2. Any body promise made to $\mathbf{pd}_1$ by $\mathbf{pd}_2$ is satisfied. If $(\mathbf{x}, \underline{d}) \in \Pi_1$, then $\downharpoonright (d_2, \Pi_2) \subseteq \underline{d}$.
3. Any head promise made to $\mathbf{pd}_2$ by $\mathbf{pd}_1$ is satisfied. If $(\Cap, \overline{d}) \in \Pi_2$, then $\overline{d} \subseteq \circlearrowleft_{\mathbf{x}} (\mathbf{pd}_1)$

5.2.2. Abstract heaps using promise-carrying data

We now have the components to build *promise-carrying data algebras*. These algebras are the analogy of structural addressing algebras (definition 42), but are built with promise-carrying data.

Definition 98 (Promise-carrying data algebra)**.** Let $(\text{StructAddrs}, \text{Data}, \text{addrs}, \text{comp})$ be some structural addressing algebra (definition 42). Let $(\text{StructAddrs}_{\Cap}, \text{Data}_{\Cap}, \text{addrs}_{\Cap}, \text{comp}_{\Cap})$ be the associated underlying data algebra (definition 90). Given a choice of promise-carrying data defined using these previous two algebras PromData, the associated **promise-carrying data algebra** is defined as:

$$(\text{StructAddrs}_P, \text{PromData}, \text{addrs}_P, \text{comp}_P)$$

where

1. The structural addresses are those of the underlying data: $\text{StructAddrs}_P = \text{StructAddrs}_{\Cap}$.
2. The addresses function is the addresses function of the underlying data, applied to the underlying data component of a promise-carrying data: $\text{addrs}(\mathbf{pd}) = \text{addrs}_{\Cap}(\mathbf{pd}{\downarrow_1})$.

3. The compression function comp_P is that defined on PROMDATA, given in definition 97.

When unambiguous, $\text{comp}_P(\mathbf{x}, \mathbf{pd}_1, \mathbf{pd}_2)$ is written $\mathbf{pd}_1 \circledast \mathbf{pd}_2$, and addrs_P is written addrs.

It is important to remember that we are dealing with *data*. These richer data are formed atop of some structural addressing algebra (STRUCTADDRS, DATA, addrs, comp) (definition 42), and so have their own notions of compression and addresses. As we did with promise-carrying data, we now define an algebra for promise-carrying data.

Definition 99 (Promise-carrying data algebra)**.** Let (STRUCTADDRS, DATA, addrs, comp) be some structural addressing algebra (definition 42). Let $(\text{STRUCTADDRS}_{\Subset}, \text{DATA}_{\Subset}, \text{addrs}_{\Subset}, \text{comp}_{\Subset})$ be the associated underlying data algebra (definition 90). Let PROMDATA be the set of promise-carrying data defined using these previous two algebras. Then, the associated *promise-carrying data algebra* is defined as:

$$(\text{STRUCTADDRS}_P, \text{PROMDATA}, \text{addrs}_P, \text{comp}_P)$$

where

1. The structural addresses are those of the underlying data: $\text{STRUCTADDRS}_P = \text{STRUCTADDRS}_{\Subset}$.
2. The addresses function is the addresses function of the underlying data, applied to the underlying data component of a promise-carrying data $\text{addrs}_P(\mathbf{pd}) = \text{addrs}_{\Subset}(\mathbf{pd}{\downarrow_1})$.
3. The compression function comp_P is that defined on PROMDATA, given in definition 97.

When unambiguous, $\text{comp}_P(\mathbf{x}, \mathbf{pd}_1, \mathbf{pd}_2)$ is written $\mathbf{pd}_1 \circledast \mathbf{pd}_2$, and addrs_P is written addrs. The data of the underlying algebra are equivalent to data with no promises $d \in \text{DATA}_{\Subset} = (d, \emptyset, \emptyset)$.

This algebra inherits many useful properties from the underlying structural addressing algebra. However, we must first ensure that the compression function preserves the well-formedness of the richer data we are using. For the lemmas, assume a structural ad-

dressing algebra (StructAddrs, Data, addrs, comp) for the promises, and the associated underlying data algebra ((StructAddrs$_P$, PromData, addrs$_P$, comp$_P$).

Lemma 18 (Compression is closed on PromData)**.** *Let* $\mathbf{pd}_1, \mathbf{pd}_2 \in$ PromData*,* $\mathbf{x} \in$ StructAddrs *and* $\mathbf{pd}_1 \circledast \mathbf{pd}_2$ *be defined. Then,* $\mathbf{pd}_1 \circledast \mathbf{pd}_2 \in$ PromData*.*

Proof. Assume $\mathbf{pd}_1 = (d_1, \Pi_1)$ and $\mathbf{pd}_2 = (d_2, \Pi_2)$. The result of $\mathbf{pd}_1 \circledast \mathbf{pd}_2$ is:

$$(d_1 \circledast d_2, (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap))$$

As per the definitions of promises and promise-carrying data (definitions 94 and 95 respectively), this will be valid promise-carrying data if: 1) there are no duplicate promise addresses; and 2) every promise is addressed to either $\Cap$ or a body address within the data.

Note first that, by the addressing properties of the underlying structural addressing algebras, the set $\text{addrs}(d_1 \circledast d_2)$ contains the addresses $\text{addrs}((d_1 \setminus \{\mathbf{x}\})) \cup \text{addrs}(d_2)$. Call this the *address containment property.*

1. Proceed by contradiction. Assume an address a that identifies two promises: $\{(a, \overline{\underline{d}}_1), (a, \overline{\underline{d}}_2)\} \subseteq (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap)$.

   Notice that $a \neq \Cap$, as by construction $\Cap$ is removed from Π_2 in the compression, and Π_1 is well-formed. Ergo $a \in$ StructAddrs $\setminus \{\Cap\}$. Therefore, by well-formedness of $\mathbf{pd}_1$ and $\mathbf{pd}_2$, $a \in \text{addrs}_P(\mathbf{pd}_1)$ or $a \in \text{addrs}_P(\mathbf{pd}_2)$. Assume $a = \mathbf{x}$. Then it cannot be duplicated, as $\mathbf{x}$ is removed from Π_1 in the union and Π_2 is well-formed. However, it is also the case that $a \neq \mathbf{x}$. This follows from the address containment properties, which show that $\text{addrs}(d_1) \cup \text{addrs}(d_2) \setminus \{\mathbf{x}\} = \emptyset$.
2. Proceed by contradiction. Assume there is a promise in $(a, \overline{\underline{d}}) \in (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap)$ such that $a \neq \Cap$ and $a \notin \text{addrs}(d_1 \circledast d_2)$. By definition, either $a \in \Pi_1$ or $a \in \Pi_2$. By the well-formedness of $\mathbf{pd}_1$ and $\mathbf{pd}_2$, either $a \in \text{addrs}(\mathbf{pd}_1)$, or $a \in \text{addrs}(\mathbf{pd}_2)$. If $a \neq \mathbf{x}$, then by the address containment property, it must be that $a \in \text{addrs}(d_1 \circledast d_2)$, which contradicts its existence.

   If $a = \mathbf{x}$, then by the construction of $(\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap)$, a addresses a promise in Π_2. Ergo, $a \in \text{addrs}(\mathbf{pd}_2)$ by well-formedness. By the address preservation property of structural addressing algebras, $a \in \text{addrs}(d_1 \circledast d_2)$, which again contradicts its existence.

□

Of the underlying data algebra properties, the following are maintained.

Lemma 19 (Value containment)**.** *Given* Data *that is the underlying structural addressing algebra for the promise-carrying data algebra, and* Values *that is the set of values underlying* Data*,* Values $\subseteq$ Data*.*

Proof. Follows from the lift of data without promises being equal to promise-carrying data, and the value containment property of the underlying data. □

Lemma 20 (Address properties)**.** *For all* $\mathbf{pd}_1, \mathbf{pd}_2 \in$ PromData *and* $\mathbf{x} \in$ StructAddrs*, if* $\mathbf{pd}_1 \circledast_{\mathbf{x}} \mathbf{pd}_2$ *is defined then:*

1. Containment*:* $\mathbf{x} \in addrs_P(\mathbf{pd}_1)$*.*
2. Non-overlap*:* $addrs_P(\mathbf{pd}_1) \cap addrs_P(\mathbf{pd}_2) \subseteq \{\mathbf{x}\}$
3. Preservation*:* $(addrs_P(\mathbf{pd}_1) \setminus \{\mathbf{x}\}) \cup addrs_P(\mathbf{pd}_2) = addrs_P(\mathbf{pd}_1 \circledast_{\mathbf{x}} \mathbf{pd}_2)$

Proof. These follow directly from the addresses function being defined in terms of the underlying addresses function, which is applied to the underlying data. □

Lemma 21 (Compression quasi-associates)**.** *Let* $\mathbf{pd}_1, \mathbf{pd}_2, \mathbf{pd}_3 \in$ PromData *and* $\mathbf{x}, \mathbf{y} \in$ StructAddrs*, with* $\mathbf{pd}_1 = (d_1, \Pi_1), \mathbf{pd}_2 = (d_2, \Pi_2), \mathbf{pd}_3 = (d_3, \Pi_3)$*. If* $\mathbf{y} \in addrs_P(\mathbf{pd}_2)$ *and either* $\mathbf{y} \notin addrs_P(\mathbf{pd}_1)$ *or* $\mathbf{y} = \mathbf{x}$*, then*
$\mathbf{pd}_1 \circledast_{\mathbf{x}} (\mathbf{pd}_2 \circledast_{\mathbf{y}} \mathbf{pd}_3) = (\mathbf{pd}_1 \circledast_{\mathbf{x}} \mathbf{pd}_2) \circledast_{\mathbf{y}} \mathbf{pd}_3$*.*

Proof. Assume that $\mathbf{x} \neq \mathbf{y}$ (the case where $\mathbf{x} = \mathbf{y}$ is similar). Assume further that the left-hand side, $\mathbf{pd}_1 \circledast_{\mathbf{x}} (\mathbf{pd}_2 \circledast_{\mathbf{y}} \mathbf{pd}_3)$ is defined. Then, it must equal:

$$\left(\; d_1 \circledast_{\mathbf{x}} (d_2 \circledast_{\mathbf{y}} d_3), \;\; (\Pi_1 - \mathbf{x}) \cup ((\Pi_2 - \mathbf{y}) \cup (\Pi_3 - \Cap)) - \Cap \; \right)$$

The removal of a promise from a set is idempotent. Therefore, we have:

$$\left(\; d_1 \circledast_{\mathbf{x}} (d_2 \circledast_{\mathbf{y}} d_3), \;\; (\Pi_1 - \mathbf{x}) \cup ((\Pi_2 - \mathbf{y} - \Cap) \cup (\Pi_3 - \Cap)) \; \right)$$

As $\mathbf{x} \neq \mathbf{y}$, there are no promises named $\mathbf{y}$ in Π_1. Moreover, as standard set union associates, we have:

$$\left(\; (d_1 \circledast_{\mathbf{x}} d_2) \circledast_{\mathbf{y}} d_3, \;\; ((\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap)) - \mathbf{y} \cup (\Pi_3 - \Cap) \; \right)$$

which is equal to $(\mathbf{pd}_1 \circledast_{\mathbf{x}} \mathbf{pd}_2) \circledast_{\mathbf{y}} \mathbf{pd}_3$ (data quasi-associativity holds by the definition of structural addressing algebras). Ergo, if the left is defined, the right is defined and equal to it. □

The proof that compression is quasi-commutative is similar. The underlying data has a simple left identity property, as $\mathbf{x} \circledast d = d$ always. The addition of promises breaks this simple identity, as data with a head promise will require additionally properties of super-data. However, there are still identities, which can be seen via construction based upon the choice of d. We show the existence of left identities below; the right identities are demonstrated in a similar manner.

Lemma 22 (Existence of left identities)**.** *For all* $\mathbf{pd} \in$ PromData *and* $\mathbf{x} \in$ StructAddrs*, there exists an* $\mathbf{pd}_{\mathbf{x}} \in$ PromData *such that* $\mathbf{pd}_{\mathbf{x}} \circledast \mathbf{pd} = \mathbf{pd}$*.*

Proof. Assume $\mathbf{pd} = (d, \Pi)$. We will construct of $\mathbf{pd}_{\mathbf{x}} = (\mathbf{x}, \Pi_{\mathbf{x}})$ by picking $\Pi_{\mathbf{x}}$ based on the contents of Π. Without futher information, the compression is:

$$(\mathbf{x} \circledast d, (\Pi_{\mathbf{x}} - \mathbf{x}) \cup (\Pi - \Uparrow))$$

Assume that $(\Uparrow, \overline{d}) \in \Pi$. In this case, we pick $\Pi_{\mathbf{x}} = \{(\mathbf{x}, \underline{d}), (\Uparrow, \overline{d})\}$. This is well-formed by construction. By calculation, using the identity property of data:

$$\begin{array}{rl} & (\mathbf{x} \circledast d, (\Pi_{\mathbf{x}} - \mathbf{x}) \cup (\Pi - \Uparrow)) \\ = & (d, (\{(\mathbf{x}, \underline{d}), (\Uparrow, \overline{d})\} - \mathbf{x}) \cup (\Pi - \Uparrow)) \\ = & (d, \{(\Uparrow, \overline{d})\} \cup (\Pi - \Uparrow)) \\ = & (d, \Pi) \end{array}$$

as required. The cases where $\mathbf{pd}$ has no head promise is similar. □

The identity properties were used by structural addressing algebras to show that the choice of hole is irrelevant. One could always compress another structural address in, so that $d \circledast \mathbf{y}$ was defined if $\mathbf{y} \notin \text{addrs}(d)$. Without the simple identity properties, we cannot prove this directly. However, we can still show that the choice of address does not matter.

Lemma 23 (Address irrelevance)**.** *For all* $\mathbf{pd}_1, \mathbf{pd}_2 \in$ PromData *and* $\mathbf{x} \in \mathit{addrs}_P(\mathbf{pd}_1)$*,* $\mathbf{y} \notin \mathit{addrs}_P(\mathbf{pd}_2)$*, if* $\mathbf{pd}_1 \circledast \mathbf{pd}_2$ *is defined, there exists* $\mathbf{pd}_{\mathbf{y}}$ *such that* $\mathbf{pd}_1 \circledast \mathbf{pd}_{\mathbf{y}} \circledcirc{\mathbf{y}} \mathbf{pd}_2 = \mathbf{pd}_1 \circledast \mathbf{pd}_2$*.*

Proof. Knowing that $\mathbf{pd}_1 \circledast \mathbf{pd}_2$, assume that $\mathbf{pd}_1 = (d_1, \Pi_1)$ and $\mathbf{pd}_2 = (d_2, \Pi_2)$. We will consider the case where $(\mathbf{x}, \underline{d}_1) \in \Pi_1$ and $(\Uparrow, \overline{d}_2) \in \Pi_2$, (ergo, $\underline{d}_2 \subseteq \underline{d}_1$, $\overline{d}_1 \subseteq \overline{d}_2$).

We construct $\mathbf{pd}_{\mathbf{x}}$ in a similar fashion to building the identities for data, by giving it the promises that match the expectations of $\mathbf{pd}_1$ and $\mathbf{pd}_2$. Pick $\mathbf{pd}_{\mathbf{y}} = (\mathbf{y}, \Pi_{\mathbf{y}})$ where $\Pi_{\mathbf{y}} = \{(\Uparrow, \overline{d}_1), (\mathbf{y}, \underline{d}_1)\}$. The result then follows by calculation, as in lemma 35. □

These properties are sufficient to demonstrate that promise-carrying data algebras are structural addressing algebras. Ergo, they can be used as the underlying method of splitting data for structural separation logic.

Corollary 1 (Promise-carrying data algebras are structural addressing algebras)**.** *All promise-carrying data algebras are structural addressing algebras.*

Proof. By lemmas 18, 31, 20 33 35 and 36, and the definition of structural separation algebras (definition 42). □

5.3. Reasoning with promises

As stated earlier, the abstract allocation relation is not a specific rule that is added to the system, but rather one deduced from the semantic consequence relation. The exact formulation used in chapter 3 no longer holds, as it is not aware of promises. However, a similar rule can be shown to exist for each use of this enriched framework. It creates and destroys promises when needed, and also repartitions those already in existence, to ensure the continued validity of each abstract heap cell. Providing a general form of this rule is not helpful, as each choice of promise will use heap cells with a notation natural to the promises being issued. We have seen examples in both the path and list indexing cases.

The assertions are similar, except that we need a notion for describing the promises associated with data. As a convention, we attach the promise to the *heap cell address* as a superscript:

$$\mathbf{a}^{P_1,P_2,\cdots} \mapsto \phi$$

Such an assertion describes a heap cell addressed by $\mathbf{a}$, containing promise-carrying data described by ϕ, where the data is associated the promises set containing promises each described by P_i. If a heap cell has no promises, we omit the superscript.

The reasoning with these abstract heaps proceeds virtually identically to that seen already in this thesis, with command axioms given using heap cells annotated with promises. However, With views p, q and r now describing promise-carrying abstract heaps, *stability* must be carefully checked. That is, if a view p contains heap cells annotated with promises, and a view r contains the heap cells with data fulfilling these promises, we must ensure that no update is allowed such that r becomes r', and r' *breaks* the promises. As with all stability, this is ensured by the standard *atomic soundness* property (parameter 12), which states that: for all axioms $(p, \mathrm{C}, q) \in \textsc{Axioms}$, with command action $[\![\mathrm{C}]\!]$

$$\forall r \in \textsc{Views}.\ [\![\textsc{c}]\!]\lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$$

When the choice of promises and command axioms means that this above can be proven for every axiom, then we say the promises are *naturally stable*. This is because they required no more machinery to use than normal structural separation logic.

Stability also governs the creation and removal of promises via abstract allocation and deallocation. Recall that abstract allocation is not something *added* to the reasoning system. Rather, it is an result of the choice of reification and *semantic consequence* relation (described in section 3.3.3). Unlike theorem 4, we cannot give a general form of abstract allocation here. This is because the validity of an allocation depends on the choice of promises for the underlying system, and the specific choices made in the allocation. A typical instance takes the form:

$$\left(\alpha^{P_1,\cdots,P_n} \mapsto \phi_1 \circledast \phi_2\right) \preccurlyeq\succcurlyeq_\Gamma \exists \beta.\ \left(\alpha^{Q_1,\cdots,Q_n,Q_m} \mapsto (\phi_1 \wedge \diamond\beta) * \beta^{R_1,\cdots,R_o} \mapsto \phi_2\right)$$

The Q_i and R_i promises will be a combination of *new* promises for the β hole, and those within P_i which have been distributed between α and β. Our inability to give a general form of the rule does not affect soundness, as it remains just a use of the sound semantic consequence rule. Uses must always be justified underlying semantic inclusion $p \preccurlyeq q \iff \forall r \in \textsc{Views}.\ \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$.

5.3.1. Axiomatising the library of trees with paths

We now show how promises allow small axioms for our tree library that uses paths. We add structural addresses to the trees to create abstract trees in the standard way, and so create *abstract rooted trees*. We also add the promise head address to paths, creating *abstract paths*.

Definition 100 (Abstract rooted trees and paths)**.** The set of **abstract trees** $\textsc{AbsTrees}$, ranged over by $\mathbf{t}, \mathbf{t}_1, \cdots, \mathbf{t}_n$, is defined inductively as follows: for all $i \in \mathbb{N}^+$, $\mathbf{x} \in \textsc{StructAddrs}$

$$\mathbf{t} ::= i[\mathbf{t}] \mid \mathbf{t}_1 \otimes \mathbf{t}_2 \mid \varnothing \mid \mathbf{x}$$

where $\otimes$ is associative and commutative with identity $\varnothing$, and both node identifiers

and body addresses are unique.

The set of **abstract rooted trees** RootedTrees, ranged over by $rt, rt_1, \cdots, rt_n$, is defined as:

$$\textsc{AbsRootedTrees} = \{\top[\mathbf{t}] \mid \mathbf{t} \in \textsc{AbsTrees}\} \cup \textsc{StructAddrs}$$

The set of **abstract tree paths** AbsTreePaths, ranged over by $\mathbf{pth}, \mathbf{pth}_1, \cdots, \mathbf{pth}_n$, is defined as: for all $i \in \mathbb{N}^+$

$$\mathbf{pth} ::= i \mid \top \mid \Cap \mid i/\mathbf{pth} \mid \top/\mathbf{pth}$$

where / is associative. Abstract tree paths are equal up to the associativity of /.

We consider a structural addresses to be an abstract rooted tree to ensure that the entire tree can be abstractly allocated if needed. We use abstract trees to create an underlying data algebra using head address $\Cap$, and define HeadData and BodyData as per definitions 90, 91 and 92.

Notice that if an abstract path contains an abstract address, it must be $\Cap$ and it must be at the end. We lift path resolution (definition 88) to abstract trees and abstract paths in the natural way. These paths state how to get directly from the top of the tree to a subtree. They are thus *head promises* only. We define promises that describe all the possible head data that resolve a given path to the head promise address. A promise-carrying data with a path promise will be assured that it will eventually compose into a tree, and be resolved to by the promised path.

Definition 101 (Path promises for linear paths)**.** The function defining head data where the head promise address $\Cap$ is at the end of path p, $\pi^{\cdot} : \textsc{Paths} \to \textsc{HeadData}$ is defined as:

$$\pi^p = \{\overline{d} \in \textsc{HeadData} \mid \forall d \in \overline{d}.\ \text{resolve}(p, d) = \Cap\}$$

The set of **promises for linear paths** are then defined as:

$$\textsc{HeadProms} = \{(\Cap, \pi^p) \mid p \in \textsc{TreePaths}\} \cup \{(\Cap, \Cap)\}$$

There are no body promises, so $\textsc{BodyProms} = \emptyset$.

$$\{\alpha^{\mathrm{P}} \mapsto \mathrm{M}[\mathrm{T} \wedge \nexists\alpha.\ \Diamond_F(\alpha) \wedge \nexists \mathrm{U},\mathrm{V}.\ (\mathrm{U} \otimes \mathrm{N} \otimes \mathrm{V})] * E \wedge (e_p \Rightarrow \mathrm{P}/\mathrm{M}) \vee (e_p \Rightarrow \mathrm{M} \wedge \mathrm{P} = \Cap \wedge \mathrm{M} = \mathrm{R})\}$$
$$\texttt{createNode}(e_p,\ e_n)$$
$$\{\alpha^{\mathrm{P}} \mapsto \mathrm{M}[\mathrm{T} \otimes \mathrm{N}[\varnothing]] * E\}$$

$$\{\alpha^{\mathrm{P}} \mapsto \mathrm{N}[\mathrm{T} \wedge \mathit{is\_complete}] * E \wedge e_p \Rightarrow \mathrm{P}/\mathrm{N}\}$$
$$\texttt{removeNode}(e_p)$$
$$\{\alpha^{\mathrm{P}} \mapsto \varnothing * E\}$$

Figure 5.10.: Axioms for tree commands using paths

The set of head promises encompasses all possible heads that resolve any path p. The unusual case $(\Cap, \Cap)$ describes the promise that the data is the entire tree; the only thing that can fit over it is an empty context.

The data-specific assertion language for abstract trees is essentially identical to that for the trees of section 61. We extend logical values with abstract paths. We can now define *promise-carrying heap cell assertions* that use promises to indicate the path at which the body address can ultimately be found.

Definition 102 (Promise-bearing linear path heap cells)**.** The two promise-bearing linear path heap cells, and their interpretations, are defined as:

$$\begin{aligned}
(\!|\mathrm{R} \mapsto \phi|\!)^{\Gamma} &= \{\mathrm{R} \mapsto (d, \emptyset)\}) \mid d \in (\!|\phi|\!)^{\Gamma}\} \\
(\!|\alpha^{\mathrm{E}} \mapsto \phi|\!)^{\Gamma} &= \begin{cases} \left\{ \mathbf{x} \mapsto (d, \Pi) \;\middle|\; \begin{array}{l} d \in (\!|\phi|\!)^{\Gamma}, p = (\!|\mathrm{E}|\!)(\Gamma), \\ \Pi = \pi^p \end{array} \right\} & \text{if } \begin{array}{l} \Gamma(\alpha) = \mathbf{x}, \\ \mathbf{x} \in \textsc{StructAddrs} \end{array} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

With these, the axioms for the tree command using paths are given in figure 5.10. They use a simple extension to logical expressions allowing the analysis of paths. Notice that the pre-conditions for the creation commands capture the entire forest being added to. This must happen to ensure the name of node being created does not clash with any existing node name. We do not use *is_complete* in these cases, but instead the "somewhere at forest level" assertion. The axioms allow body addresses deeper within the sub-tree, as we are *only* checking to ensure the uniqueness of the new sibling's name.

These axioms are atomically sound with respect to a natural choice for their actions. The proofs follow the same pattern demonstrated before in this thesis. The only interesting

change is the restrictions on possible completions, and additional stability requirements induced by promises. We sketch the case for `removeNode`.

Lemma 24 (Atomic soundness of `removeNode`)**.** *The command* `removeNode(`e_p`)` *is atomically sound.*

Proof. We must first provide an atomic action for the `removeNode(`e_p command. For convinience, we define this using abstract trees via the $\Cap$ address: for all $s \in$ TreeHeaps and $e_p \in$ Paths

$$[\![\texttt{removeNode(}e_p\texttt{)}]\!](s) = \begin{cases} \{s[\mathrm{R} \mapsto \mathbf{t}_1 \Cap \varnothing\} & \text{if } s(\mathrm{R}) = rt, rt = \mathbf{t}_1 \Cap \mathbf{t}_2, \mathrm{resolve}([\![e_p]\!](s), \mathbf{t}_1) = \Cap \\ \text{undefined} & \text{otherwise} \end{cases}$$

Consider the pre-condition for `removeChild`, $\alpha^{\mathrm{P}} \mapsto \mathrm{N}[\mathrm{T} \wedge \mathit{is\_complete}] * E \wedge e_p \Rightarrow \mathrm{P}/\mathrm{N}$. By the definition of assertion interpretation, this interprets as the view $\mathbf{x} \mapsto (n[\mathbf{t}], \pi^p) * e$ for some $\mathbf{x}$, n, and $\mathbf{t}$ where $\mathrm{addrs}(\mathbf{t}) = \emptyset$, and path p, where the evaluation of $e_p = p/n$, and e is arbitrary variable resource. The key use of the promise is to restrict the set of reifications of this resultant view. Because, by definition, $\mathbf{x}$ will only compress into super-data that place $\mathbf{x}$ at path p, and completions added by the reification process will be forced to place $\mathbf{x}$ at path $\mathbf{x}$. Therefore, for all $r \in$ Views:

$$\lfloor \mathbf{x} \mapsto (n[\mathbf{t}], \pi^p) * e * r \rfloor = \{\mathrm{R} \mapsto rt\} \sqcup \mathit{varHeap}$$

where the sub-tree $n[\mathbf{t}]$ is found in rt by resolving path e_p, and $\mathit{varHeap}$ is an arbitrary heap of variables that can evaluate e_p. This is exactly the safe resource for running the `removeNode(`e_p`)` command. It updates this heap to $\{\mathrm{R} \mapsto rt'\} \sqcup \mathit{varHeap}$, where rt' is identical to rt except that the path e_p now resolves to a directory without an N node, the sub-tree it refers to having been removed. By interpreting the post-condition, we see this must always be contained in the reification of $\alpha^{\mathrm{P}} \mapsto \varnothing * E$.

The only subtle step in seeing this is that that any choice of r that composed with pre-condition the will still compose with the post-condition. We must ensure that no r can have been promised a fact that is no longer true. Fortunately, we have chosen the axiom to ensure that, regardless of the choice of r, it is stable. The only possible promises invalidated by this command are those regarding paths that encounter N during resolution. If such a promise has been issued, the body address it has been issued to must be within the sub-tree at e_p. However, the axiom ensures that no such body addresses exist (via *is_complete*), so no promises can have been broken. □

5.3.2. List indexing

We now turn to axiomatising the `item` command for the lists library. We first formalise the promises we will use.

Definition 103 (Promises for index lists)**.** The **list index head data with respect to x** and the **list length body data with respect to x** are defined as:

$$\begin{aligned} \Leftarrow i &= \{\overline{d} \mid d \in \overline{d}, d = [pl_1 \otimes \mathbf{x} \otimes pl_2], |pl_1| = i\} \\ \Rightarrow i &= \{\underline{d} \mid d \in \underline{d}, |d| = i\} \end{aligned}$$

The **promises for the list library** are defined as:

$$\textsc{Promises} = \begin{array}{ll} & \bigcup_{i \in \mathbb{N}} \{(\Cap, \Leftarrow i)\} \quad \text{List index promises} \\ \cup & \bigcup_{i \in \mathbb{N}, \mathbf{x} \in \textsc{StructAddrs}} \{(\mathbf{x}, \Rightarrow i\} \quad \text{List length promises} \end{array}$$

Here, we are using more than just head promises. Many promises may accrue as abstract allocation occurs. To ease notation, we define logical expressions for denoting sets of promises.

Definition 104 (Logical expressions for list promises)**.** The logical expressions for the list library with item command and associated interpretation function are extended with the following expressions:

$$\begin{aligned} &\llbracket \emptyset \rrbracket(\Gamma) && \triangleq \emptyset \\ &\llbracket \textsc{a} \Leftarrow \mathrm{E} \rrbracket(\Gamma) && \triangleq \begin{cases} \Gamma(\textsc{a}) \Leftarrow \llbracket \mathrm{E} \rrbracket(\Gamma) & \text{if } \Gamma(\textsc{a}) \in \textsc{StructAddrs}, \llbracket \mathrm{E} \rrbracket(\Gamma) \in \mathbb{N} \\ \text{undefined} & \text{otherwise} \end{cases} \\ &\llbracket \textsc{a} \Rightarrow \mathrm{E} \rrbracket(\Gamma) && \triangleq \begin{cases} \Gamma(\textsc{a}) \Rightarrow \llbracket \mathrm{E} \rrbracket(\Gamma) & \text{if } \Gamma(\textsc{a}) \in \textsc{StructAddrs}, \llbracket \mathrm{E} \rrbracket(\Gamma) \in \mathbb{N} \\ \text{undefined} & \text{otherwise} \end{cases} \\ &\llbracket \mathrm{E}_1, \cdots, \mathrm{E}_n \rrbracket(\Gamma) && \triangleq \{\llbracket \mathrm{E}_1 \rrbracket(\Gamma), \cdots, \llbracket \mathrm{E}_n \rrbracket(\Gamma)\} \end{aligned}$$

We then define a promise-carrying abstract list heap cell assertion.

Definition 105 (Promise-carrying heap assertion)**.** The abstract heap assertions of the list library are extended to promise-carrying data with:

$$(\!|\mathrm{A}^{\mathrm{E}_P} \mapsto \phi|\!)^\Gamma = \left\{ \mathbf{a} \mapsto (d, \Pi) \;\middle|\; \begin{array}{l} \mathbf{a} = \Gamma(\mathrm{A}), d \in (\!|\phi|\!)^\Gamma, \\ \Pi = (\!|\mathrm{E}_P|\!)(\Gamma), \end{array} \right\}$$

The assertion $\mathrm{A}^{\emptyset} \mapsto \phi$ is written $\mathrm{A} \mapsto \phi$.

The abstract heap construction using these data is similar to the tree case of section 3.3.5. More interesting is the abstract allocation relation. The enriched notion of compression and data means it can now be used to *pick* which promises will be issued at point of allocation. The following are all valid uses, which follows directly from the definitions of compression and abstract heap construction:

$$\mathrm{L} \mapsto [1 \otimes 2 \otimes 3] \quad \rightsquigarrow_\Gamma \quad \exists \alpha.(\mathrm{L} \mapsto [1 \otimes \alpha \otimes 3] * \alpha \mapsto 2$$
$$\mathrm{L} \mapsto [1 \otimes 2 \otimes 3] \quad \rightsquigarrow_\Gamma \quad \exists \alpha.(\mathrm{L}_{\alpha \Leftarrow 1} \mapsto [1 \otimes \alpha \otimes 3] * \alpha^{\Uparrow \Leftarrow 1} \mapsto 2)$$

$$\exists \alpha.(\mathrm{L}_{\alpha \Leftarrow 1} \mapsto [1 \otimes \alpha \otimes 3] * \alpha^{\Uparrow \Leftarrow 1} \mapsto 2)$$
$$\rightsquigarrow_\Gamma$$
$$\exists \alpha, \beta.(\mathrm{L}^{\beta \Rightarrow 1}_{\alpha \Leftarrow 1} \mapsto [\beta \otimes \alpha \otimes 3] * \alpha^{\Uparrow \Leftarrow 1} \mapsto 2 * \beta_{\Uparrow \Rightarrow 1} \mapsto 1)$$

$$\exists \alpha, \beta.(\mathrm{L}^{\beta \Rightarrow 1, \alpha \Rightarrow 1}_{\alpha \Leftarrow 1} \mapsto [\beta \otimes \alpha \otimes 3] * \alpha^{\Uparrow \Leftarrow 1}_{\Uparrow \Rightarrow 1} \mapsto 2 * \beta_{\Uparrow \Rightarrow 1} \mapsto 1)$$
$$\rightsquigarrow_\Gamma$$
$$\exists \alpha, \beta, \gamma.(\mathrm{L}^{\beta \Rightarrow 1, \alpha \Rightarrow 1}_{\alpha \Leftarrow 1, \gamma \Leftarrow 2} \mapsto [\beta \otimes \alpha \otimes \gamma] * \alpha^{\Uparrow \Leftarrow 1}_{\Uparrow \Rightarrow 1} \mapsto 2 * \beta_{\Uparrow \Rightarrow 1} \mapsto 1 * \gamma^{\Uparrow \Leftarrow 2} \mapsto 3)$$

The axiom for the list indexing command is given below. It uses the index head promise to ensure it is returning the correct element. The length of J is checked to ensure we are capturing only a single list element in J.

$$\{\alpha^{\Uparrow \Leftarrow \mathrm{I}} \mapsto \mathrm{J} * \mathtt{j} \to \text{-} * E \wedge e \Rightarrow \mathrm{I} \wedge |\mathrm{J}| = 1\}$$
$$\mathtt{j := item}(e)$$
$$\{\alpha^{\Uparrow \Leftarrow \mathrm{I}} \mapsto \mathrm{J} * \mathtt{j} \to \mathrm{J} * E\}$$

For ensure these list index promises are naturally stable, we must ensure no command can break them. The only mutative commands, and thus the only commands that could break the promise, are `append` and `remove`. The `append` command evidently cannot break a promise; it only ever extends the *end* of a list, so cannot perturb the index of any body address. However, the `remove` command *could* alter such an index, by removing an element earlier in the list.

To restore natural stability, we simply alter `remove` command to mirror `append`: only the last element of a list can be removed. The new axiom is thus:

$$\{\mathrm{L} \mapsto [\alpha \otimes \mathrm{I}] * E \wedge e \Rightarrow \mathrm{I}\}$$
$$\texttt{remove}(e)$$
$$\{\mathrm{L} \mapsto [\alpha] * E\}$$

This is a strong restriction, reducing the list to a stack, and showing the limits of naturally stable promises. In chapter 7, we restore full list behaviour by introducing *obligations*.

5.4. Summary

We have demonstrated *promises*, a method for recording additional contextual information about structured data that has been abstractly allocated. Promises allow us to consider more libraries than are possible with pre-existing techniques alone, by contributing:

1. **Smaller axioms for commands**: Promises allow even smaller command axioms than the techniques shown in previous chapters, as seen by the list commands of section 5.3.2. This allows more natural specifications of the behaviours of the commands, and furthers the number of safe concurrent programs that can be proven.
2. **Passive access to structured data**: Promises act as *passive* data; information about the state that cannot be mutated. They are thus act similarly to a permission system, but instead of allowing read-only access to the entirety of a datum (thus rendering it immutable), allow *properties* of that datum to be shared instead.
3. **Local reasoning for paths**: Promises allow the assertions demonstrating the *position* of some data in a structure to be handled separately from the data itself. This is most evident with the paths example, section 5.3.1, where the data at a path can be separated from a structure, whilst the nodes along a path need not be. This allows multiple sub-data to be separated at once, and enables true local reasoning for libraries using paths. This is not possible in previous work on context logic.

6. POSIX file systems

Much of the content in this chapter is joint work with Gian Nizk.

This chapter uses structural separation logic with promises to present a novel axiomatic semantics for the *POSIX file system library*. The primary difficulty with file system reasoning is that most commands access files via *paths*. Rather than accessing resource via a globally unique identifier, these paths are *resolved* with respect to the entire file-system tree. Local reasoning is therefore normally difficult, as each command has a (potential) footprint of the entire tree. We will use promises to show how, despite paths, structural separation logic allows local reasoning about file systems.

File systems are an abstraction between the storage mechanisms of a machine, and the programs that run on that machine. Regardless of the particular hardware used, a file system provides users with the intuitive notion of data stored in a tree-like hierarchy of directories and files. There are several choices of file system abstraction, the two dominant being those of Windows and of POSIX. We focus on the POSIX interfaces as they are both ubiquitous (present in all major operating systems either by default or as an add-on) and comprehensively standardised in [1].

This standardisation of POSIX, whilst detailed, is presented mostly in English. It is complex and contains several subtleties that programmers can miss. Computer users often experience problems caused by programmers incorrectly using a file system API. Mistakes are common in client programs. For example, without care, software installers can make assumptions about file system structures that are not true, causing them to fail and leave the programs they are installing unusable. This chapter attempts focuses on these problems by providing a specification that allows the verification of client code. This is distinct from most formal systems for file systems, which focus on the verification of implementations.

The POSIX specification, even the sub-part dealing with file systems, is too large to consider at once. We therefore provide a compositional specification of a *core subset* of the POSIX file system, distilling from the large specification a set of data structures and commands that captures the essence of the library, whilst being as faithful as possible to the

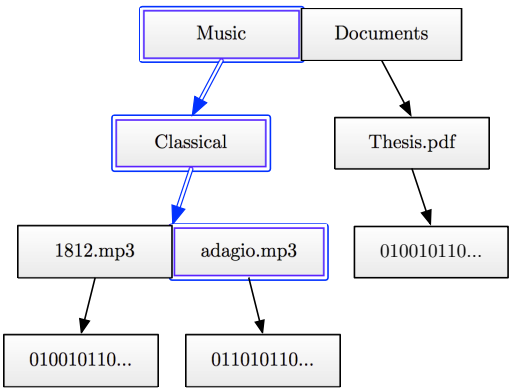


Figure 6.1.: An example file-system tree, with the linear path $\top/Music/Classical/adagio.mp3$ highlighted.

standard. We term our subset *featherweight POSIXFS*. We provide an imperative machine for featherweight POSIXFS, focusing on the *file system*, which intuitively represents the contents of the hard disk, and the *process heap*, which represents the contents of the heap memory used by programs accessing the file system. We exclude advanced features such as asynchronous IO and sockets, to focus on the fundamental structure of *files* within trees of *directories*. We select a subset of commands relevant to our fragment. This subset is *closed*, in that the commands within it do not create structures outside our model, and *natural*, in that the commands are regularly used in file system programs.

Our fragment initially considers only *linear paths*. These paths take the shortest route to a file during resolution, matching the inductive structure of the directory tree exactly. An example file-system tree is given in figure 6.1, with the path /Music/Classical/adagio.mp3 highlighted. We use promises to axiomatise the commands that identify files via paths, and so demonstrate natural small axioms for our command subset.

We use featherweight POSIXFS to verify a *software installer* example, using structural separation logic to reason locally about updating a widget directory. Software installers provide a common class of client programs for file systems. Newly obtained software is typically provided as a *bundle*, downloaded or copied onto the user's file system. The goal of a software installer is take this bundle and place the contents at the correct points in the user's file system such that the software can run. Installers also perform other tasks, such as removing previous versions of the software and purging incompatible files that may have been generated by it. We verify an example software installer, demonstrating that it behaves sensibly: if the installer fails, the file system will be unchanged; if it succeeds,

the program will be successfully installed; and there will be no other outcomes.

Whilst there has been substantial work on the formal specification of file systems, it has mostly focused on *refinement*, giving specifications that are suitable for creating verified implementations. Our work most closely mirrors the Z specification of given by Morgan and Suffrin [50]. This work predates the POSIX standardisation, and has been the basis of most formal work since. For example, based on these specifications, [30] used Z notation to formally specify a set of operations covering primitive file-manipulation functionality, mechanising them in the Z/Eves theorem prover, and refining them to an implementation in Java. This work was part of an effort to build a verified POSIX file system [32], in response to the mini-verification challenge by Joshi and Holzmann [48] to verify a file system associated with a Mars lander.

An alternative approach is given in [41], formalizing an abstract file system by means of partial functions from paths to data. However, this work does not focus on modelling a standardised file system such as POSIX, but rather a more generic file store. As with Z specifications, the main motivation is to verify implementations rather than client programs.

One system verifying the *usage* of file system libraries is *Forest* [28], a Haskell domain-specific language for describing and manipulating file stores. Forest uses classical tree logics to define the semantics of a core calculus for file-system tree shapes, creating a declarative and type-based file system programming paradigm. Again, this is not a specification for a standardised library like POSIX.

6.1. Distilling featherweight POSIXFS

The POSIX standard is very large, with issue 7 of the base specifications spanning nearly four thousand pages. It covers many different types of operating system abstraction, of which the file system is just one. The prose is a mix of English descriptions for the formal concepts, C language source code to describe some programming interfaces, and English rationales for certain decisions. Though rigorously defined, the standard is not presented in a way amenable to formal analysis. We therefore first analyse the standard to extract *featherweight POSIXFS*, a tractable subset, focusing on file systems, which we can formalise for analysis. In forming this subset, we refer to many definitions within the standard [1].

6.1.1. Pruning the file-system tree

Despite being presented as a tree by most user interfaces, the file-system defined by POSIX is actually a *directed graph*. The nodes of the graph are the seven types of *file*, enumerated in standard reference 3.164. Despite the name, these are not what are generally thought of as files, but instead are just *"objects that can be written to, or read from, or both"*. Each file has a unique *file serial number* (standard reference 3.175), commonly called an *inode*. The graph edges are given by *directory files* (reference 3.129). Each directory file contains an unordered list of *directory entries* (reference 3.130). Each entry is a named *hard link* (reference 3.191) to another file, and the same file may have multiple incoming hard links. The result is a *file system hierarchy* (reference 4.5). One directory file is distinguished as the *root directory* (reference 3.324). The overall result is a graph similar to that of figure 6.2, using inode 1 as the root.

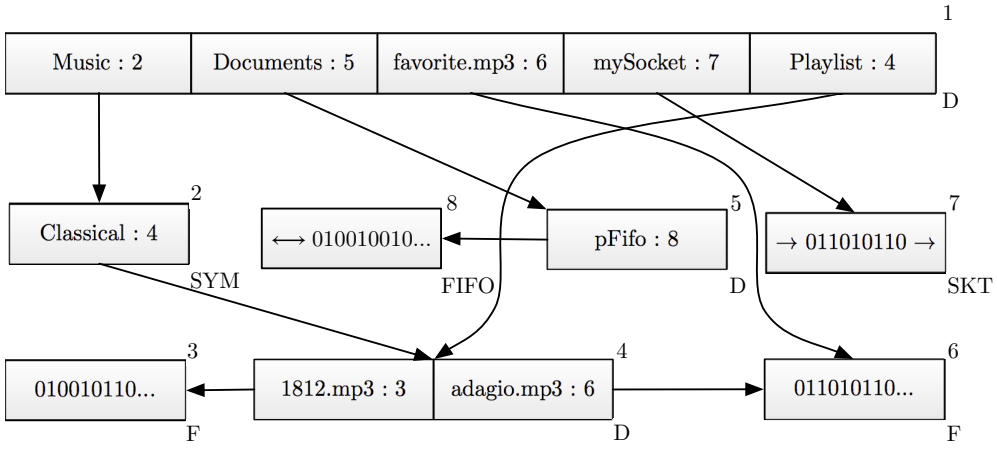


Figure 6.2.: An example file-system, as permitted by the POSIX specification.

Of the seven types of file, we will initially model just *regular files* (labelled by the subscript F in the diagram) and *directories* (labelled D). We will outline a possible formalisation of *symbolic links* in chapter 7. The other types of file (block special, character special, FIFO, and socket) are less commonly used. We believe that they can be added to our subset without difficulty if needed. By removing these file types from consideration, we simplify file systems to those shaped like figure 6.3.

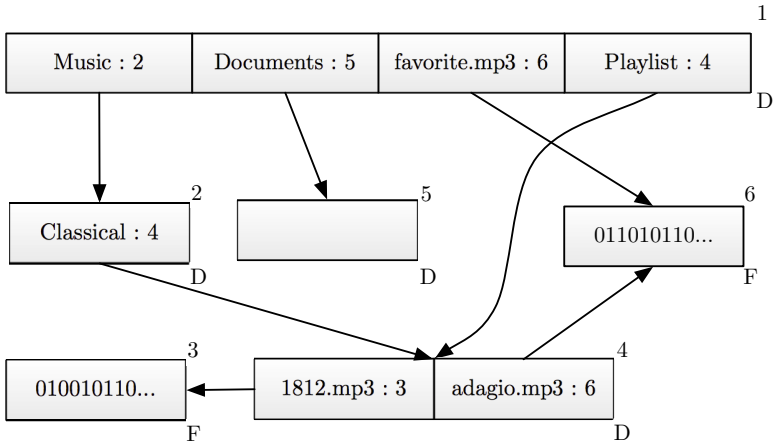


Figure 6.3.: An example file system, considering only regular files and directories.

Figure 6.3 more closely resembles the common intuition of a file-system structure. The directory files consist of file names referring either other directories or regular files. The *regular files* contain byte sequences. However, there may be many hard links to the same file. This is typically desirable, as many different directories may want to reference the same file. Unfortunately, if hard links point to *directories* rather than regular files, *cyclic* file systems can be created. Whilst this is allowed by POSIX, it is disallowed by almost every POSIX implementation due to the complexities of file systems with "loops" in their structure. We restrict our system so that hard links reference regular files only.

With this restriction, a file system can be seen as a tree of directories. Within this tree structures, *files* are referenced (possibly many times) by their inodes, resulting in a heap of files. We will represent the directory tree using an inductive data structure. Our subset will not permit programs to *open* directories as if they were general files (an operation that is rare). We therefore remove their inodes from this structure without affecting the usefulness of our fragment, and obtain a file-system tree like figure 6.4.

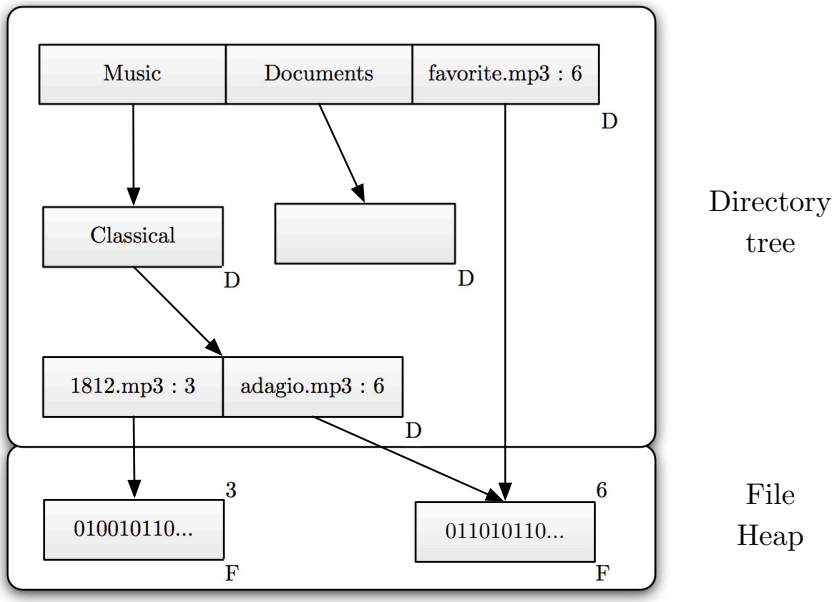


Figure 6.4.: An example file system, considering only trees of directories, with regular files at the fringe.

Formalisation

We work with file systems under the restrictions above. We first formalise the file-system tree. Two general sets of data are necessary for this.

Parameter 21 (File systems data for trees)**.** Assume a set of **file names** for directory entries FNAMES (reference 3.170), ranged over by a, b, c, with a distinguished **root name** $\top \in$ FNAMES. Assume also a countably infinite set of **inode identifiers** INODES (reference 3.175), ranged over by $\iota, \kappa, \cdots$.

To maximise the number of POSIX implementations we can specify, we do not impose a limit on the length of file names. Therefore, the POSIX constant `NAME_MAX`, which defines the system limit on name length, is infinite in our setting. We now define the set of *directories*. Directories can be defined as an inductive data type, representing the trees of directory files (reference 3.129), each containing zero or more directory entries (reference 3.130). We choose our representation by referencing the standard's specification of the *internal directory structure* (section "format of directory entries" in `dirent.h`). This

gives us free reign to pick a representation of directories that is as general as the standard, as long as we can determine the inode and name for each directory entry. Our subset does not give inodes to directories, so entries in the directory tree are either other *named directories* or *named file links to an inode*.

Definition 106 (Unrooted directories)**.** The set of **unrooted directories** UDIRS, ranged over by $ud, ud_1, \cdots, ud_n$, is defined as: for all $a \in (\text{FNAMES} \backslash \{\top\}), \iota \in \text{INODES}$:

$$\begin{array}{rcll} ud & ::= & \varnothing & \text{Empty directory} \\ & & \mid a : \iota & \text{File-link entry named } a \\ & & \mid a[ud] & \text{Directory entry named } a \\ & & \mid ud_1 + ud_2 & \text{Directory composition} \end{array}$$

where $+$ is commutative and associative with identity $\varnothing$, and the directory entries have sibling unique names. Unrooted directories are equal up to the properties of $+$.

We also define *rooted directories*, which represent the single starting point of the file system hierarchy. Together, rooted and unrooted directories form the set of general directories.

Definition 107 (Rooted directories)**.** The set of **rooted directories** RDIRS, ranged over by $rd, rd_1, \cdots, rd_n$, is defined as:

$$\text{RDIRS} \triangleq \{\top[ud] \mid ud \in \text{UDIRS}\}$$

The set of **directories** DIRS, ranged over by $d, d_1, \cdots, d_n$ is defined as:

$$\text{DIRS} \triangleq \text{UDIRS} \cup \text{RDIRS}$$

Every entry within the file-system tree is typed as either a directory or regular file. A file-link entry, $a : \iota$, has type F. A directory entry $a[ud]$ has type D. We also give a type to entries that do not exist, N. This will be used to represent files that cannot be found in a directory tree.

Definition 108 (Entry types)**.** The set of **entry types** DETYPES is defined as:

$$\text{DETYPES} = \{\text{F}, \text{D}, \text{N}\}$$

We define the *inodes function* to extract the set of used inodes from a directory. This function will be useful in checking well-formedness of our structures.

Definition 109 (Inodes function)**.** The **inodes function** inodes : DIRS $\to$ $\mathcal{P}$(INODES) is defined by induction on directories to collect the set of inodes used. It is similar to definition 37.

Rooted directories form the file-system tree. The heap of regular files consist of inode identifiers pointing to *byte sequences* (reference 3.317). Byte sequences are lists of *bytes* (reference 3.84).

Definition 110 (Bytes)**.** The set of **bytes** BYTES, ranged over by $b, b_1, \cdots, b_n$, is defined as the integers in the $[0, 255]$ interval. The set of **byte sequences** BYTESEQS, ranged over by $bs, bs_1, \cdots, bs_n$, is defined as: for all $b \in$ BYTES

$$bs ::= b \mid b \cdot bs \mid \varnothing$$

where $\cdot$ is associative with identity $\varnothing$. Byte sequences are equal up to the properties of $\cdot$.

Definition 111 (Byte sequence length)**.** The **byte sequence length** function $|\cdot|$: BYTESEQS $\to \mathbb{N}$ is defined as the length of the input byte sequence, and is standard.

We now define a *file-system heap* as the structure which represents a file system. It consists of a rooted directory and a heap of regular files. This heap must contain at least the files referenced by the file-link entries in the file-system tree.

Definition 112 (File-system heaps)**.** Let $\mathcal{F}$ be the **directory tree address**, where $\mathcal{F} \notin$ INODES. The set of **file systems** FILESYSTEMS, ranged over by $fs, fs_1, \cdots, fs_n$, is defined as:

$$\text{FILESYSTEMS} \triangleq \left\{ fs \;\middle|\; \begin{array}{l} fs : (\{\mathcal{F}\} \to \text{RDIRS}) \sqcup (\text{INODES} \xrightarrow{\text{fin}} \text{BYTESEQS}), \\ \text{inodes}(fs(\mathcal{F})) \subseteq \text{dom}(fs) \end{array} \right\}$$

If a regular file is not referenced by any directory entries, and is not currently in use, the standard says that it will be destroyed. We choose not to model this destruction of regular files by commands, instead assuming that file-system heaps are garbage collected. This allows our presentation to focus on the core behaviours of commands, rather than resource cleanup.

6.1.2. Paths

Commands that access the file-system tree identify the data they work on via *pathnames* (POSIX standard [1] reference 3.266, more often simply called *paths*). A path is a list of *pathname components* (reference 3.267) separated by the '/' character. A pathname component is either a directory or regular file name, the reserved name '.' or the reserved name '..' (reference 3.169).

As our subset allows neither hard links to directories nor symbolic links (for the moment), paths consisting of filenames match the inductive structure of the file-system tree. They are *resolved* in the tree, with each component indicating the next sub-tree for resolution, and the final component indicating the target. However, paths containing the '.' and '..' characters are *not* inductively related to the tree. Whereas a normal pathname component states the next directory or file in the path, the '.' name indicates a hard link to the directory that contains it. The '..' name is a hard link to the *parent* of the directory. In POSIX, all directories must contain '.' and '..'. As we do not currently allow hard links to directories, we do not consider '.' or '..'. Our paths are thus *linear*, taking the shortest route through the file-system tree to the entry they identify.

Linear paths are either *relative* or *absolute*. Relative paths can be used to resolve an entry in any part of the tree. Absolute absolute paths always start at the root. POSIX gives this root directory the path '/'. For typesetting clarify, we label the root path '/' as $\top$. Ergo, absolute linear paths are of the form $\top$/list/of/file/names.

Definition 113 (Linear Paths)**.** The set of **relative linear paths** RelativePaths, ranged over by $rp, rp_1, \cdots, rp_n$, is defined as: for all $a \in$ FNames

$$rp ::= a \mid rp/a$$

The set of **absolute linear paths** AbsolutePaths, ranged over by $ap, ap_1, \cdots, ap_n$, is defined as:

$$\textsc{AbsolutePaths} = \{\top\} \cup \{\top/rp \mid rp \in \textsc{RelativePaths}\}$$

The set of **linear paths** Paths, ranged over by $p, p_1, \cdots, p_n$, is defined as $\textsc{Paths} = \textsc{RelativePaths} \cup \textsc{AbsolutePaths}$

Paths locate entries within a directory tree via a process of *path resolution* (standard reference 4.12). Path resolution starts at the root of the file system, and follows each path component down one level of the structure to find the entry. The resolution of linear paths matches the inductive structure of the file system hierarchy. Paths do not always resolve in a given directory tree, corresponding to the file not being found.

Definition 114 (Path resolution)**.** The **path resolution function** resolve : $\textsc{Paths} \to \textsc{Dirs} \rightharpoonup \textsc{Dirs}$ is defined as: for all $a \in \textsc{FNames}, rp \in \textsc{RelativePaths}, d, d_1, d_2 \in \textsc{Dirs}$

$$\begin{array}{lcl}
\text{resolve}(a, d + a : \iota) & = & a : \iota \\
\text{resolve}(a, d_1 + a[d_2]) & = & a[d_2] \\
\text{resolve}(\top, \top[d]) & = & \top[d] \\
\text{resolve}(a/rp, d_1 + a[d_2]) & = & \text{resolve}(rp, d_2) \\
\text{resolve}(\top/rp, \top[d]) & = & \text{resolve}(rp, d) \\
\text{otherwise} & & \text{undefined}
\end{array}$$

We will need to *concatenate* paths to perform certain operations.

Definition 115 (Path concatenation)**.** The **path concatenation function** $pathConcat : \textsc{Paths} \to \textsc{Paths} \rightharpoonup \textsc{Paths}$ is defined as:

$$\begin{array}{ll}
pathConcat(p, a) & = p/a \\
pathConcat(p_1, a/p_2) & = pathConcat(p_1/a, p_2)
\end{array}$$

where, if the pathname component $\top$ would appear in the result as anything other than the first component, the result is undefined. The path concatenation $pathConcat(p_1, p_2)$ is written p_1/p_2.

6.1.3. Process heap

The heaps of definition 112 represent the state of the storage device holding the file system. We will also need *runtime* structures representing normal heap memory, and specific file-system data held by programs. We call this the *process heap*. The process heap consists of the structures created by opening files and enumerating directories, along with normal heap memory.

An *open file description* (reference 3.253) is a record holding information about the regular files opened by a program. This record contains the inode of the file, and the offset within the file that read and write operations will begin from. Open file descriptors are created by the `open` commands, used by `read`, `write`, and `lseek` and are destroyed by the `close` command (all given in definition 119).

Definition 116 (Open file description)**.** The set of **open file descriptions** OpenFileDescriptions, ranged over by $fd, fd_1, \cdots, fd_n$, is defined as:

$$\text{OpenFileDescriptions} \triangleq \text{INodes} \times \mathbb{N}$$

Programs almost never open directories as if they were files. Instead, they enumerate the contents with special directory management commands. All such interaction with directories is via *directory streams* (reference 3.131). These represent the state of an open directory in the form of a *entry set*, recording the entries that were in a directory when it was opened. They are used to support the `opendir`, `readdir` and `closedir` commands (all given in definition 119).

Definition 117 (Directory streams)**.** The set of **directory streams** OpenDirStreams, ranged over by $ds, ds_1, \cdots, ds_n$ is defined as:

$$\text{OpenDirStreams} \triangleq \mathcal{P}(\text{FNames})$$

The *process heap* is a structured heap holding open file descriptions and directory streams. These structures are referred to via *open file descriptors* (reference 3.166)[1]. The heap also includes a standard heap, as commands that read and write to files use standard heap memory as the destination and source of these commands.

[1]Not to be confused with open file descrip*tions*, definition 116.

Definition 118 (Process heap)**.** Assume a countably infinite set of **open file descriptors** OpenFileDescriptors, ranged over by $n, m, \cdots$. The set of **process heaps** Ph, ranged over by $\mathbf{ph}, \mathbf{ph}_1, \cdots, \mathbf{ph}_n$, is defined as:

$$\textsc{Ph} \triangleq \left\{ \mathbf{ph} \;\middle|\; \mathbf{ph} : \begin{array}{l} \textsc{OpenFileDescriptors} \xrightarrow{\text{fin}} \left(\begin{array}{c} \textsc{OpenDirStreams} \\ \cup \\ \textsc{OpenFileDescriptions} \end{array} \right) \\ \sqcup \quad \mathbb{N}^+ \xrightarrow{\text{fin}} \mathbb{N} \end{array} \right\}$$

6.1.4. Selection of commands

The POSIX standard defines over 160 commands that manipulate the file system. Not all of these commands are relevant to the set of file types we model. Moreover, many of the commands are *redundant*, in that they can be implemented using the other commands. We pick a *core command* set that is: *closed*, in that it allows all the expected manipulations of the file system structure we define; and *natural*, in that the commands we choose are those typically used by programs.

We classify our core commands into *structural* commands that manipulate the file-system tree, *primitive IO* commands that read and write the contents of files in the heap, *state* commands for querying file attributes, and *heap* commands that allocate and deallocate linear blocks of normal heap memory.

Definition 119 (Core POSIX fragment)**.** The set of **core POSIX commands** FSCommands consists of the **structural commands** StructCommands, ranged over by $\textsc{c}_{STR}$, **IO commands** IOCommands, ranged over by $\textsc{c}_{IO}$, **state commands** StateCommands, ranged over by $\textsc{c}_{STA}$, and **regular heap commands** HeapCommands, ranged over by $\textsc{c}_{H}$. These are defined as: for all program expressions `path`, `existing`, `new`, `flags`, `dir`, `flags`, `fd`, `size`, `buffer`, `offset`, `whence` and program variables `ndir`, `nfd`, `nsize`, `noffset`, `s` and `ptr`

```
C_STR ::=  mkdir(path) | rmdir(path)
         | link(existing, new) | unlink(path)
         | rename(existing, new)

C_IO ::=   ndir := opendir(path)
         | fn := readdir(dir) | closedir(dir)
         | fd := open(path, flags)
         | c := read(fd, buf, sz)
         | c := write(fd, buf, sz)
         | noffset := lseek(fd, offset, whence)
         | close(fd)

C_STA ::=  s := stat(path)

C_H ::=    ptr := malloc(size)
           free(ptr)

C_FS ::=   C_STR | C_IO | C_STA | C_H
```

Structural commands

Structural commands are those concerned with manipulating the directory tree. They never access the file heap nor the process heap.

`mkdir(path)` When `path` evaluates to p/a, this command creates a new empty directory named a inside the directory resolved to by p. If p does not resolve, or p/a does resolve, the command faults.

`rmdir(path)` Removes the empty directory resolved to by `path`. If `path` does not resolve, resolves to a directory with children, or resolves to the root directory, the command faults.

`link(existing, new)` When the value of `existing` is p_1/a and that of `new` is p_2/a, `link` creates a file-link entry a in the directory named by p_2 linking the same inode as p_1/a. If p_1 or p_2 do not resolve to existing directories, if p_1/a resolves to a directory, or if p_2/a resolves at all, the command faults.

`unlink(path)` Removes the file link resolved to by `path`. If `path` does not resolve to a file link, the command faults.

`rename(existing, new)` The command moves the directory entry resolved to by `existing` so it becomes the directory entry resolved to by `new`. This happens through the following cases:

1. Renaming a file to a new file: If `existing` resolves to a file link to inode ι, `new` is of the form p/a and p resolves to a directory without an entry named a, then the directory entry resolved to by `existing` is removed and a new file-link entry named by a is created, resolved to by `new`, referencing inode ι.
2. Renaming a file to the same name as an existing file: If `existing` resolves to a file link to inode ι and `new` resolves to any file link, then the directory entries resolved to by `existing` and `new` are removed and a new file-link entry is created, resolved to by `new`, referencing inode ι.
3. Renaming a directory to a new directory: If `existing` resolves to a directory that is not an ancestor of `new`, `new` is of the form p/a and p resolves to a directory without an entry named a, then the directory entry resolved to by `existing` is removed and a new directory type entry is created, resolved to by `new`, with the same contents as that originally resolved to by `existing`.
4. Renaming a directory to the same name as an existing directory: If `existing` resolves to a directory that is not an ancestor of `new` and `new` resolves to a directory with no children, then the directory entries resolved to by `existing` and `new` are removed and a new directory type entry is created, resolved to by `new`, with an identical form to that originally resolved to by `existing`.
5. Renaming something to itself: If `new` and `existing` have the same path value, and it resolves to some entry, then `rename` does not affect the file system.

In all other cases, `rename` faults.

Input/Output commands

These commands are concerned with the contents of files. Whilst some examine the directory tree to set some initial resource state, most access just the file and process heaps.

`ndir := opendir(path)` Creates a fresh directory stream in the process heap containing the names of the entries within the directory resolved to `path` and assigns the address

of the stream to `ndir`. If `path` does not resolve to a directory, the command faults.

`fn := readdir(dir)` Returns an entry from an open directory stream. When the process heap has a directory stream entry for `dir`, then:

1. when the directory stream is non-empty, non-deterministically selects and removes an entry, assigning the value to `fn`;
2. when it is empty, assigns ϵ (the empty file name) to `fn`.

If `dir` does not evaluate to a directory stream address or the open directory heap does not have an entry for `dir`, the command faults.

`closedir(dir)` Removes the open directory stream referenced by `dir` from the open directory heap. If `dir` is not a directory stream address or the open directory heap does not have an entry for `dir`, the command faults.

`nfd := open(path, flags)` Either opens or creates a new file.

1. If the directory entry resolved to by `path` is of the form $a : \iota$, `open` creates a fresh file descriptor (say, n) in the open file heap associated with the open file description $(\iota, 0)$, and assigns n to `nfd`. Additionally, if `flags` is `O_TRUNC`, this command sets the contents of file ι in the file heap to be the empty byte sequence.
2. If `path` does not resolve, is of the form p/a, p resolves to a directory, and `flags` = `O_CREAT`, then: a new entry is added to the file heap with a fresh inode (say, ι) and the empty byte sequence as contents; and, a file-link entry named a is created in the directory resolved by p linking the inode ι. Then `open` proceeds as in case 1.

If `flags` is not 0, `O_TRUNC` or `O_CREAT`, `path` resolves to a directory type entry, or `path` is of the form p/a such that p does not resolve, the command faults.

`c := read(fd, buff, size)` If `fd` refers to a file descriptor with inode ι and offset *offset* in the file heap, this command reads from file ι at most `size` bytes, starting at *offset*, into the heap buffer pointed to by `buff`. If *offset* + `size` bytes exceeds the length of the file, reads only until the end. Assigns to `c` the number of bytes read into the buffer. The command updates the file offset by adding the length of the byte sequence actually read. If the open file heap has no entry for `fd`, `size` is not a natural number, or if the buffer pointed to by `buff` is not of at least length `size`, the command faults.

`c := write(fd, buff, size)` If `fd` refers to a file descriptor with inode ι and offset *offset*, writes `size` bytes from the heap buffer `buff` to the file associated with ι, starting at *offset*, assigning `size` to `nsize`. Any contents present in the file between the offset and the length of `str` are overwritten, and the file length is extended if needed. If the file offset is greater than the length of the file, then the command first writes as many 0 bytes to the file as needed to make the length of the contents equal to the file offset. The file offset is updated by adding `size`. If the process heap has no entry for `fd`, `buff` is not a heap buffer of at least length `size`, the command faults.

`noffset := lseek(fd, offset, whence)` Moves the file offset for an open file descriptor. If the open file heap maps `fd` to an entry (ι, o), and the file heap maps ι to an entry with contents $\boldsymbol{s}$ then `lseek` changes the open file heap mapping for `fd` to (ι, o') and `noffset` is set to o', where o' calculated as:

1. if `whence` is `SEEK_SET`, o' is `offset`;
2. if `whence` is `SEEK_CUR`, o' is `offset` $+ o$;
3. if `whence` is `SEEK_END`, o' is `offset` $+ |\boldsymbol{s}|$.

Note that the command is allowed to set the new file offset to a value greater than the length of the file. If o' is negative, `whence` is none of `SEEK_SET`, `SEEK_CUR` and `SEEK_END`, or the open file heap has no mapping for `fd`, then `lseek` faults.

`close(fd)` Closes the open file descriptor `fd`, by removing it from the process heap. If the open file heap does not contain an entry `fd`, the command faults.

Status command

Our single status command just examines the directory tree. We distinguish this from a structural command because it does not modify the file system.

`t := stat(path)` If the directory entry resolved to by `path` is a file-link entry, then assigns the file type constant F to `t`. If it is of directory type, assigns D to `t`. If the path `path` does not resolve in the tree, assigns the file type N.

Heap commands

The two heap commands allocate and deallocate linear blocks of heap memory. The commands use a *malloc block head* which stores the length of the block that has been

allocated. This block head is a single integer stored in the address just below that of the memory that is returned.

`ptr := malloc(size)` If `size` evaluates to a positive integer $size$, allocates $size + 1$ fresh contiguous heap cells starting at address ptr. At address ptr, stores $size$. In addresses $[ptr + 1, ptr + size]$, stores non-deterministic bytes. Assigns $ptr + 1$ to `ptr`. Faults if the evaluation of `size` is not a positive integer.

`free(ptr)` : If `ptr` evaluates to a positive integer ptr, and the heap contains a positive integer $size$ at address $ptr - 1$, deallocates from the heap all cells in the range $[ptr - 1, ptr + size]$. Faults if `ptr` is not a positive integer, $ptr - 1$ is not a positive integer, or the heap does not contain a cell for every address in the $[ptr - 1, ptr + size]$ range.

Our core POSIX commands are simplified versions of the true POSIX commands, but retain the behaviours pertinent to our subset. We have adapted the specifications to work with a simple imperative language, eliding many artifacts of C. We omit error return codes and treat any errors as faults[2]. Our command descriptions correspond to the behaviour specified by POSIX that is relevant to our file system model choices. The only exceptions are the `readdir` command, and the `malloc` and `free` commands, which we simplify. As specified by POSIX, `readdir` is non-deterministic:

> *"If a file is removed from or added to the directory after the most recent call to* `opendir` *..., whether a subsequent call to* `readdir` *returns an entry for that file is unspecified."*

Therefore, mutations made to a directory that is opened via *opendir* can be reflected in calls to `readdir` in one of three ways:

1. All changes are reflected, so that files that have been added to the directory are returned and files that have been removed are not.
2. Some changes are reflected, so that if files are added or removed, whether they will be returned is unknown.
3. No changes are returned. The state of the directory as it was when `opendir` was used is always the state used.

[2]We have studied errors, and include them in an upcoming paper based upon this chapter. However, they draw the focus away from the core specification and use of structural separation logic. We thus omit them here.

Ideally, our model would capture the full non-determinism of `readdir`. However, the first two choices raise troubling questions about behaviour. To simplify this initial presentation, we pick the third option: `readdir` will return the elements that were present in the directory at the point it was initially opened with `opendir`.

The POSIX standard for `malloc(size)` and `free` states that malloc should return the starting address of a block of memory of length `size`. It also states that, when this address is passed to `free`, the entire block should be deallocated. To enable this, we must store the length of the block, so that `free` can ensure the entire block is removed. Moreover, the standard states that if an address *not* returned by `malloc` is passed to `free`, the behaviour is undefined.

We give a valid specification for `malloc` and `free`, but do not capture the entire non-determinism of the standard. Specifically, we *implement* the behaviour by storing the length of a block in the heap, at the address immediately below the returned heap block. This, in theory, enables programs using our choices to obtain deterministic behaviour not allowed by the standard (such as manually resizing blocks of memory). To capture the full spirit of POSIX here, we would require a more abstract representation of heap memory returned by `malloc`. As this is not the focus of the chapter, we choose the simpler, less general, behaviour.

6.2. Imperative machines for featherweight POSIXFS

Having selected the featherweight POSIXFS subset, we now develop an imperative machine for programming with it. This machine is formed of a heap containing variables, a file system heap (definition 112), and a process heap (definition 118). The large range of types used by POSIX means the selection of variable values is much richer than our previous examples. We have seen each of the types before, except the *empty file name* constant ϵ. This will be used by commands which would normally return a file name, but cannot due to the state of the file system.

Definition 120 (Variable values)**.** Let $\epsilon \notin \text{FNAMES}$ be the **empty filename**. The set of **variable values** PVALS, ranged over by $v, v_1, \cdots, v_n$ is defined as:

$$\begin{aligned} \text{PVALS} \triangleq \quad & \mathbb{Z} \sqcup \{\textbf{true}, \textbf{false}\} \sqcup \text{FNAMES} \sqcup \{\epsilon\} \sqcup \text{INODES} \sqcup \text{DETYPES} \\ & \sqcup \text{BYTESEQS} \sqcup \text{PATHS} \sqcup \text{OPENFILEDESCRIPTORS} \end{aligned}$$

Some commands are parameterised with values that are not addresses or paths. For instance, the `lseek` command takes a parameter `whence` instructing it in what direction to seek. Following POSIX, we define *symbolic constants* to capture these parameters. The constants are `SEEK_SET`, `SEEK_CUR`, `SEEK_END` (for `lseek`) and `O_TRUNC` (for `open`). We will assume that these constants exist in any variable store (corresponding to, for example, them being set up by a runtime or included in some global definitions file).

We enrich the standard set of program expressions with expressions for *path manipulation*. This includes the POSIX commands `basepath` and `dirpath`, which determine the final path component and the prefix leading to the final path component respectively. In POSIX, these are commands. However, as they do not mutate the heap, it we give them as expressions.

Definition 121 (Program expressions)**.** The set of **program expressions** Exprs, ranged over by $e, e_1, \cdots, e_n$, is extended from that of example 5 by:

$e ::=$	$e_1 \cdot e_2$	Byte sequence concatenation
	$\mid \lvert e \rvert$	Byte sequence length
	$\mid e_1/e_2$	Path concatenation
	$\mid \texttt{basepath}(e) \mid \texttt{dirpath}(e)$	Path analysis

The evaluation of expressions is mostly standard. We give only the cases for path analysis.

Definition 122 (Evaluation of path expressions)**.** The expression evaluation function $(\!|\cdot|\!)(\cdot)$ for the expressions of definition 121 is standard, except for: for all e such that for all e_1, e_2 with $(\!|e_1|\!)(s), (\!|e_2|\!)(s) \in$ Paths:

$$
\begin{array}{rll}
(\!|e_1/e_2|\!)(s) & = (\!|e_1|\!)(s)/(\!|e_2|\!)(s) & \text{if } (\!|e_2|\!)(s) \notin \textsc{AbsolutePaths} \\
(\!|\texttt{basepath}(e)|\!)(s) & = a & \text{if } (\!|e|\!)(s) = p/a \\
(\!|\texttt{dirpath}(e)|\!)(s) & = p & \text{if } (\!|e|\!)(s) = p/a \\
(\!|\texttt{basepath}(e)|\!)(s) & = \top & \text{if } (\!|e|\!)(s) = \top \\
(\!|\texttt{dirpath}(e)|\!)(s) & = \top & \text{if } (\!|e|\!)(s) = \top
\end{array}
$$

The structured heaps for featherweight POSIXFS fuse a variable store, a file system and a process heap.

Definition 123 (Structured heaps)**.** The set of **structured heaps** for POSIXFS PFSHeaps, ranged over by $fsh, fsh_1, \cdots, fsh_n$, are defined as:

$$\text{PFSHeaps} \triangleq \{fsh \mid fsh : (\text{PVars} \xrightarrow{\text{fin}} \text{PVals}) \sqcup \text{FileSystems} \sqcup \text{Ph}\}$$

6.2.1. Command actions

In this chapter, we will not prove the soundness of our reasoning with respect to an underlying set of command actions[3]. Omitting the actions does not weaken the rigour of our argument, as the POSIX standard itself does not provide a set of operational semantics. Ergo, any set we create will have to be created by us, and justified against the English prose. We are also responsible for creating the small axioms. Any mismatch between the two would therefore be a surprise, as we are writing the operational semantics purely to justify the axiomatic semantics!

When a library is specified without a pre-existing formal operational semantics, we feel that giving only the axioms more closely fits our library reasoning agenda. This is because we are not trying to capture a specific operational implementation, but the most general *specification* that we can give to the library commands. Soundness of the reasoning should instead be justified both with respect to the prose of the library document, but also against *implementations* of the library both via refinement techniques and empirical testing. We will discuss our ongoing work with refinement in the conclusions of chapter 8.

Comment 11. Using actions and axioms, as I did in DOM, is useful when there is a preexisting formalism against which one can check our choices (e.g. [65] for DOM). It is also useful when presenting work to audiences who may be very familiar with operational semantics, but less so with axiomatic. Presenting both allows an easier route to understanding for some people.

6.3. Reasoning

We now define the reasoning for featherweight POSXIFS. We proceed as in previous chapters, first introducing the data types needed to build the abstract heap, then creating an assertion language, and finally presenting the small axioms.

[3]Recall that atomic command actions, parameter 6, give the operational semantics of commands.

6.3.1. Abstract file systems

The only rich data type in the file systems presentation is the directory structure. We add structural addressing to this using contexts to create *abstract directories*. Notice that we will not need structural addresses for the process heap (definition 118), as these values have a flat structure.

Definition 124 (Abstract directories)**.** The set of **abstract unrooted directories** AbsUDirs, ranged over by $\mathbf{ud}, \mathbf{ud}_1, \cdots, \mathbf{ud}_n$, is defined as: for all $a \in (\text{FNames} \setminus \{\top\}), \mathbf{x} \in \text{StructAddrs}$

$$\mathbf{ud} ::= \varnothing \mid a : \iota \mid a[\mathbf{ud}] \mid \mathbf{ud}_1 + \mathbf{ud}_2 \mid \mathbf{x}$$

where $+$ is associative and commutative with identity $\varnothing$, directory entries are sibling unique, and body addresses are unique. Abstract unrooted directories are equal up to the properties of $+$.

The set of **abstract rooted directories** AbsRDirs, ranged over by $\mathbf{rd}, \mathbf{rd}_1, \cdots, \mathbf{rd}_n$, is defined as:

$$\text{AbsRDirs} \triangleq \{\top[\mathbf{ud}] \mid \mathbf{ud} \in \text{AbsUDirs}\} \cup \text{StructAddrs}$$

The set of **abstract directories** AbsDirs, ranged over by $\mathbf{d}, \mathbf{d}_1, \cdots, \mathbf{d}_n$, is defined as:

$$\text{AbsDirs} \triangleq \text{AbsUDirs} \cup \text{AbsRDirs}$$

Allowing a structural address to be used as an abstract rooted directory ensures that the *entire* directory tree can be abstractly allocated if needed.

The addresses function and composition for this data are defined in the standard manner.

Definition 125 (Addresses function for directories)**.** The *addresses function* $\text{addrs} : \text{AbsDirs} \to \mathcal{P}(\text{StructAddrs})$ is defined by induction to extract the set of structural addresses present in an abstract directory. It is similar to 37.

Definition 126 (Abstract directories composition)**.** The **abstract directories composition** function $\text{comp} : \text{StructAddrs} \to \text{AbsDirs} \to \text{AbsDirs} \rightharpoonup \text{AbsDirs}$ is

defined as substitution of a directory entry for the given structural address, if the result is contained within AbsDirs. It is similar to 38.

This reasoning for featherweight POSIXFS will use path promises in a similar manner to the example of section 5.1.1. As we use linear paths, these promises state the linear path from which the data associated with the promise was allocated. As we are using linear paths, which take the shortest route from the top of the file system tree to the data, these promises will be *head promises*, given in terms of paths that lead to $\Cap$. Along with the standard paths We use an *empty abstract path* $\varnothing_p$, which represents that $\Cap$ is at the very root of the tree. This will describe the situation when the entire file-system tree has been abstractly allocated.

Definition 127 (Abstract paths)**.** Given the set of absolute paths AbsolutePaths (definition 113), the set of **abstract paths** AbstractPaths, ranged over by $\mathbf{p}, \mathbf{p}_1, \cdots, \mathbf{p}_n$, is defined as:

$$\textsc{AbstractPaths} = \{\mathbf{p}/\Cap \mid \mathbf{p} \in \textsc{AbsolutePaths}\} \cup \{\varnothing_p/\Cap\}$$

We lift path resolution to abstract paths on abstract directory trees in the natural way. The only non-evident case is for the path $\varnothing_p/\Cap$, which only resolves as $\text{resolve}(\varnothing_p/\Cap, \Cap) = \Cap$.

Definition 128 (Promises for file systems)**.** Given a structural addressing algebra formed of abstract directories and the associated composition and addresses function, we construct the underlying data algebra $(\textsc{StructAddrs}_{\Cap}, \textsc{Dirs}_{\Cap}, \text{addrs}_{\Cap}, \text{comp}_{\Cap})$ as per definition 90. Then, the **at path** head data $@ : \textsc{AbstractPaths} \to \mathcal{P}(\textsc{AbsRDirs})$ is defined as:

$$@p = \{\mathbf{rd} \in \textsc{AbsRDirs} \mid \text{resolve}(p, \mathbf{rd}) \text{ defined}, \text{addrs}(\mathbf{rd}) = \{\Cap\}\}$$

The set of HeadProms for file systems is a set of rooted directories in which an abstract path resolves.

$$\textsc{HeadProms} \triangleq \{(\Cap, @p) \mid p \in \textsc{AbstractPaths}\}$$

There are no body promises: $\textsc{BodyProms} \triangleq \emptyset$. The set of promise sets Promises is formed as per definition 94.

Notice that these head promises always refer to a *directory* into which either another directory or file link can be composed. These promises are naturally stable with respect to the file system commands, for the same reason that the path promises were self stable in section 5.1.1. With these "at path" promises, we define the abstract heap for file systems. Notice that the process heap is unchanged from the imperative machine definition. Process heaps are entirely flat, and require no structural addressing. Notice we use the data and structural addresses from the underlying data algebra.

Definition 129 (Pre-abstract heap)**.** The set of **pre-abstract heaps** for POSIXFS PFSHeaps, ranged over by $\mathbf{fsh}, \mathbf{fsh}_1, \cdots, \mathbf{fsh}_n$, are defined as:

$$\textsc{AbsPFSHeaps} \triangleq \left\{ fsh \;\middle|\; fsh : \begin{array}{ll} & (\textsc{PVars} \xrightarrow{\text{fin}} \textsc{PVals}) \\ \sqcup & (\{\top\} \rightharpoonup (\textsc{AbsRDirs}_{\Subset} \times \textsc{Promises})) \\ \sqcup & \textsc{Ph} \\ \sqcup & (\textsc{StructAddrs}_{\Subset} \xrightarrow{\text{fin}} (\textsc{AbsDirs}_{\Subset} \times \textsc{Promises})) \end{array} \right\}$$

The set of abstract heaps for featherweight POSIXFS is those pre-abstract heaps with completions which satisfy the path promises.

6.3.2. Assertion language

As in previous chapters, we define the assertion language by giving logical values, logical expressions, data specific assertions and, finally, abstract heap assertions. The logical values and expressions for file system reasoning are richer than previous examples. This due to the variety of data types we are manipulating, and the amount of analysis needed in certain axioms (mostly focusing on paths and sets). Paths will be used to denote the path promise on a cell. Ergo, we allow the abstract "empty path" $\varnothing_p$ to enable the abstract allocation of the entire file system tree.

Definition 130 (Logical values)**.** The set of **logical values** LVals are defined as follows:

$$\begin{array}{rl} \textsc{LValues} \triangleq & \textsc{PVals} \cup \mathcal{P}(\textsc{PVals}) \cup \textsc{AbsDirs} \cup \mathcal{P}(\textsc{AbsDirs}) \\ & \cup\ \textsc{Paths} \cup \{\varnothing_p\} \cup \mathcal{P}(\textsc{AbsolutePaths}) \cup \{\varnothing_p\} \cup \textsc{StructAddrs} \end{array}$$

The logical expressions allow the standard set of logical expressions, plus normal set operations, byte sequence analysis, and path analysis. The later two are used for axiomatising the `read` and `write` commands, and the structural commands respectively. We allow sets of program values and directories to enable logical variables to store *sets* of entries, which is useful for showing invariants of certain directory structures.

Definition 131 (Logical expressions and evaluation)**.** The set of **logical expressions** LExprs, ranged over by $\mathrm{E}, \mathrm{E}_1, \cdots, \mathrm{E}_n$, is defined inductively as: for all $V \in \mathrm{PVals}, \mathrm{L}, \mathrm{L}_1, \mathrm{L}_2 \in \mathrm{LVars}$

$$
\begin{array}{lll}
\mathrm{E} ::= & V \mid \mathrm{L} \mid \mathrm{E}_1 \in \mathrm{E}_2 \mid \mathrm{E}_1 \cup \mathrm{E}_2 \mid \mathrm{E}_1 \setminus \mathrm{E}_2 & \text{Variables literals, and sets} \\
& \mid \mathrm{E}_1 \cdot \mathrm{E}_2 \mid |\mathrm{E}| & \text{Byte sequence concat. and length} \\
& \mid \mathrm{E}_1/\mathrm{E}_2 & \text{Path concatenation} \\
& \mid \texttt{basepath}(\mathrm{E}) \mid \texttt{dirpath}(\mathrm{E}) & \text{Path analysis}
\end{array}
$$

The **logical expression evaluation function** $\llbracket\cdot\rrbracket(\Gamma)\cdot : \mathrm{LEnvs} \to \mathrm{LExprs} \rightharpoonup \mathrm{LVals}$ is defined as:

$$
\begin{array}{ll}
\llbracket V \rrbracket(\Gamma) & \triangleq V \\
\llbracket \mathrm{L} \rrbracket(\Gamma) & \triangleq \Gamma(\mathrm{L}) \\
\llbracket E_1 \in E_2 \rrbracket(\Gamma) & \triangleq \begin{cases} \textbf{true} & \text{if } \llbracket E_1 \rrbracket(\Gamma) \in \llbracket E_2 \rrbracket(\Gamma), \llbracket E_2 \rrbracket(\Gamma) \text{ is a set} \\ \textbf{false} & \text{if } \llbracket E_1 \rrbracket(\Gamma) \notin \llbracket E_2 \rrbracket(\Gamma), \llbracket E_2 \rrbracket(\Gamma) \text{ is a set} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\llbracket E_1 \cup E_2 \rrbracket(\Gamma) & \triangleq \begin{cases} \llbracket E_1 \rrbracket(\Gamma) \cup \llbracket E_2 \rrbracket(\Gamma) & \text{if } \llbracket E_1 \rrbracket(\Gamma), \llbracket E_2 \rrbracket(\Gamma) \text{ are sets} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\llbracket E_1 \setminus E_2 \rrbracket(\Gamma) & \triangleq \begin{cases} \llbracket E_1 \rrbracket(\Gamma) \setminus \llbracket E_2 \rrbracket(\Gamma) & \text{if } \llbracket E_1 \rrbracket(\Gamma), \llbracket E_2 \rrbracket(\Gamma) \text{ are sets} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\llbracket \mathrm{E}_1 \cdot \mathrm{E}_1 \rrbracket(\Gamma) & \triangleq \llbracket \mathrm{E}_1 \rrbracket(\Gamma) \cdot \llbracket \mathrm{E}_2 \rrbracket(\Gamma) \\
\llbracket |\mathrm{E}| \rrbracket(\Gamma) & \triangleq |\llbracket \mathrm{E} \rrbracket(\Gamma)| \\
\llbracket \mathrm{E}_1/\mathrm{E}_2 \rrbracket(\Gamma) & \triangleq \llbracket \mathrm{E}_1 \rrbracket(\Gamma)/\llbracket \mathrm{E}_2 \rrbracket(\Gamma) \\
\llbracket \texttt{basepath}(\mathrm{E}) \rrbracket(\Gamma) & \triangleq \texttt{basepath}(\llbracket \mathrm{E} \rrbracket(\Gamma)) \\
\llbracket \texttt{dirpath}(\mathrm{E}) \rrbracket(\Gamma) & \triangleq \texttt{dirpath}(\llbracket \mathrm{E} \rrbracket(\Gamma))
\end{array}
$$

The set of data assertions are the lift of the file system data type to the assertion language, plus the required assertions of structural separation logic.

Definition 132 (File system data assertions)**.** The set of **file-system tree assertions** FSTreeAssts, ranged over by $\phi, \phi_1, \cdots, \phi_m$, are defined as:

$\phi ::=$	$\varnothing$	Empty entry
	$\mid \mathrm{E}[\phi]$	Directory type entry
	$\mid \mathrm{E}_1 : \mathrm{E}_2$	File type entry
	$\mid \phi_1 + \phi_2$	Entry list
	$\mid \top[\phi]$	File system root
	$\mid \mathrm{E}$	Logical expression
	$\mid \phi_1 \circledcirc \phi_2$	Address separation
	$\mid \diamond\alpha$	Address availability
	$\mid \phi_1 \Rightarrow \phi_2$	Implication
	$\mid \mathbf{false}$	Falsity
	$\mid \exists \mathrm{x}.\phi$	Existential

As we did in satisfaction relation for DOM (definition 83), we define the meaning of file-system tree assertions relationally.

Definition 133 (File-system tree assertion interpretations)**.** The **file-system tree assertions interpretation relation**, $\vDash \subset (\text{LEnvs} \times \text{AbsDirs}_{\Subset}) \times \text{FSTreeAssts}$, with elements $((\Gamma, \mathbf{d}), \phi)$ written $\Gamma, \mathbf{d} \vDash \phi$ is defined as:

$$
\begin{array}{rrcl}
\Gamma, \mathbf{d} \vDash & \varnothing & \iff & \mathbf{d} = \varnothing \\
\Gamma, \mathbf{d} \vDash & E[\phi] & \iff & \exists \mathbf{d}_1.\ \mathbf{d} = (\!|E|\!)(\Gamma)[\mathbf{d}_1] \wedge \Gamma, \mathbf{d}_1 \vDash \phi \\
\Gamma, \mathbf{d} \vDash & \mathrm{E}_1 : \mathrm{E}_2 & \iff & \mathbf{d} = (\!|\mathrm{E}_1|\!)(\Gamma) : (\!|\mathrm{E}_2|\!)(\Gamma) \\
\Gamma, \mathbf{d} \vDash & \phi_1 + \phi_2 & \iff & \exists \mathbf{d}_1, \mathbf{d}_2.\ \mathbf{d} = \mathbf{d}_1 + \mathbf{d}_2 \wedge \Gamma, \mathbf{d}_1 \vDash \phi_1 \wedge \Gamma, \mathbf{d}_2 \vDash \phi_2 \\
\Gamma, \mathbf{d} \vDash & \top[\phi] & \iff & \exists \mathbf{d}_1.\ \mathbf{d} = \top[\mathbf{d}_1] \wedge \Gamma, \mathbf{d}_1 \vDash \phi \\
\Gamma, \mathbf{d} \vDash & \mathrm{E} & \iff & \mathbf{d} = (\!|\mathrm{E}|\!)(\Gamma) \\
\Gamma, \mathbf{d} \vDash & \phi_1 \mathbin{@} \phi_2 & \iff & \exists \mathbf{d}_1, \mathbf{x}, \mathbf{d}_2.\ \begin{array}{l} \Gamma(\alpha) = \mathbf{x} \wedge \mathbf{d} = \mathbf{d}_1 \circledast \mathbf{d}_2 \wedge \\ \Gamma, \mathbf{d}_1 \vDash \phi_1 \wedge \Gamma, \mathbf{d}_2 \vDash \phi_2 \end{array} \\
\Gamma, \mathbf{d} \vDash & \diamond\alpha & \iff & \Gamma(\alpha) = \mathbf{x} \wedge \mathbf{x} \in \mathrm{addrs}(\mathbf{d}) \\
\Gamma, \mathbf{d} \vDash & \phi_1 \Rightarrow \phi_2 & \iff & (\Gamma, \mathbf{d} \vDash \phi_1) \implies (\Gamma, \mathbf{d} \vDash \phi_2) \\
\Gamma, \mathbf{d} \vDash & \mathbf{false} & \iff & \text{never} \\
\Gamma, \mathbf{d} \vDash & \exists \mathrm{x}.\phi & \iff & \exists x.\ \Gamma[\mathrm{x} \mapsto x], \mathbf{d} \vDash \phi
\end{array}
$$

These assertions satisfy the requirements of data specific assertions by construction. With them, we define a set of useful derived assertions.

Definition 134 (Derived Assertions)**.** The set of **derived assertions** for file systems reasoning are defined as:

$$
\begin{array}{rcl}
\Diamond\phi & \triangleq & \exists\alpha.\ (\mathbf{true} \mathbin{@} \phi) \\
\mathit{entry}(\mathrm{A}) & \triangleq & \mathrm{A}[\mathbf{true}] \vee \exists \mathrm{I}.\ (\mathrm{A} : \mathrm{I}) \\
\mathit{top\_entry}(\mathrm{A}) & \triangleq & \mathbf{true} + \mathit{entry}(\mathrm{A}) \\
\mathit{top\_address} & \triangleq & \exists\alpha.\alpha \in \textsc{StructAddrs} \wedge\ (\mathbf{true} + \alpha) \\
\mathit{names}(\mathrm{S}) & \triangleq & \forall \mathrm{A}.\ (\mathrm{A} \in \mathrm{S} \iff \mathbf{true} + \mathit{entry}(\mathrm{A})) \\
\underset{i}{\odot} b & \triangleq & (i = 0 \wedge \varnothing) \vee \left(i > 0 \wedge b \cdot \left(\underset{i-1}{\odot} b \right) \right)
\end{array}
$$

The assertion $\Diamond\phi$ is the *somewhere in the tree* assertion, stating that ϕ holds at some level of the tree structure. The assertion $\mathit{entry}(\mathrm{A})$ describes an entry in the file system; a file or directory named A. The assertion $\mathit{top\_entry}(\mathrm{A})$ states that somewhere in a list of directory entries is the entry named A (either as a file link, or a directory); it is used by e.g. `mkdir` to ensure the new directory name does not already exist. Asserting $\mathit{top\_address}$ is similar, but states there is some structural address present; that is, some children may be

missing due to abstract allocation. The assertion $\mathit{names}(\mathrm{S})$ describes that every name in the set S is present as an entry. Finally, the assertion $\bigodot_i b$ describes *repeated byte sequence composition*, repeating the byte sequence b exactly i times.

We now define the abstract heap assertions, using distinct assertions for each type of data. Notice that only abstract heap cells are annotated with promises, as the root of the file system heap is always at the root path. The annotations on each abstract address indicate the path at which they will compress into their super-data.

Definition 135 (Abstract heap cells)**.** The set of **abstract heap assertions** are those of definition 55, extending with the following additional assertions:

$$
\begin{array}{rrl}
P ::= & \top \mapsto \phi & \text{Root of file-system tree} \\
| & \alpha^{\mathrm{E}} \mapsto \phi & \text{Abstract directory cell} \\
| & \mathrm{x} \overset{\mathrm{F}}{\mapsto} \mathrm{E} & \text{Regular file Cell} \\
| & \mathrm{x} \overset{\mathrm{PH}}{\mapsto} \mathrm{E} & \text{File descriptor heap cell} \\
| & \mathrm{x} \mapsto \mathrm{E} & \text{Regular heap cell}
\end{array}
$$

The assertion interpretation function is defined as:

$$
\begin{aligned}
(\!|\top \mapsto \phi|\!)^{\Gamma} &= \{\top \mapsto (\mathbf{rd}, \emptyset) \mid \mathbf{rd} \in (\!|\phi|\!)^{\Gamma}\} \\
(\!|\alpha^{\mathrm{E}} \mapsto \phi|\!)^{\Gamma} &= \begin{cases} \{\mathbf{x} \mapsto (\mathbf{d}, (\Cap, @\mathbf{p}/\Cap) \mid \mathbf{d} \in (\!|\phi|\!)^{\Gamma}\} & \text{if } \Gamma(\alpha) = \mathbf{x}, \mathbf{x} \in \textsc{StructAddrs}_{\Cap}, \\ & \quad (\!|\mathrm{E}|\!)(\Gamma) = \mathbf{p}, \\ & \quad \mathbf{p} \in \textsc{AbsPaths} \cup \{\varnothing_p\} \\ \emptyset & \text{otherwise} \end{cases} \\
(\!|\mathrm{x} \overset{\mathrm{F}}{\mapsto} \mathrm{E}|\!)^{\Gamma} &= \begin{cases} \{\iota \mapsto (\!|\mathrm{E}|\!)(\Gamma)\} & \text{if } \Gamma(\mathrm{x}) = \iota, \iota \in \textsc{INodes} \\ \emptyset & \text{otherwise} \end{cases} \\
(\!|\mathrm{x} \overset{\mathrm{PH}}{\mapsto} \mathrm{E}|\!)^{\Gamma} &= \begin{cases} \{ofd \mapsto (\!|\mathrm{E}|\!)(\Gamma)\} & \text{if } \Gamma(\mathrm{x}) = ofd, ofd \in \textsc{OpenFileDescriptors} \\ \emptyset & \text{otherwise} \end{cases} \\
(\!|\mathrm{x} \mapsto \mathrm{E}|\!)^{\Gamma} &= \begin{cases} \{x \mapsto (\!|\mathrm{E}|\!)(\Gamma)\} & \text{if } \Gamma(\mathrm{x}) = x, x \in \mathbb{N}^{+} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

To axiomatise memory allocation, we require an inductive predicate representing an contiguous block of regular heap memory.

Definition 136 (n-cells heap predicate)**.** The **n-cells heap predicate** *ncells* is defined as:

$$ncells(start, len, data) \triangleq \exists b, bs. \begin{pmatrix} (len = 0 \wedge data = \varnothing * \mathrm{emp}) \\ \vee \\ len > 0 \wedge data = b \cdot bs * \\ start \mapsto b * ncells(start + 1, len - 1, bs) \end{pmatrix}$$

6.3.3. Axioms

The axioms for featherweight POSIXFS are given in figure 6.5. There are several common themes across the axioms.

Expression resource: The assertion E appears in almost all the assertions, and is used in conjunction with the expression mapping assertion $\Rightarrow$ to evaluate expressions.

Mixing of flat and structured data: All commands working on the file system directory tree use addresses bearing path promises. However, there are many commands operating on the process heap that deal only with *flat* resource. Structural separation logic allows the mixing of both.

Path analysis: Many POSIX commands identify resource by a path expression passed to the command. This expression must be broken down into the path components used by individual entries. This is achieved with two predicates:

$$\begin{array}{ll} path(\mathtt{path}, \mathrm{P}, \mathrm{B}, \mathrm{A}) & \triangleq \mathtt{path} \Rightarrow \mathrm{P}/\mathrm{B}/\mathrm{A} \vee (\mathtt{path} \Rightarrow \top/\mathrm{A} \wedge \mathrm{P} = \varnothing_p \wedge \mathrm{B} = \top) \\ path(\mathtt{path}, \mathrm{P}, \mathrm{A}) & \triangleq \mathtt{path} \Rightarrow \mathrm{P}/\mathrm{A} \end{array}$$

The two *path* predicates analyse a program expression, determining the path components that comprise it. The name is overloaded, with the first $path(\mathtt{path}, \mathrm{P}, \mathrm{B}, \mathrm{A})$ describes a program expression `path` that breaks down into a prefix path P, followed by a directory entry B and then entry A. The assertion allows P to be *empty*, so that $\mathrm{B} = \top$ and the path describes a root entry in the file system. This predicate is typically used to describe an entry a that does *not* exist, so that we need to know the target directory b in which it will be created. The second $path(\mathtt{path}, \mathrm{P}, \mathrm{A})$ is typically used to identify an existing file a at path P.

Checking for an existing file: Any axiom that will be adding a new directory entry must ensure the name does not clash with that of an existing file. To ensure this, we must capture all elements of the directory and ensure there are no body addresses. We achieve this with the $can\_create(a)$ predicate:

$$can\_create(a) \triangleq \neg top\_entry(a) \wedge \neg top\_address$$

This describes directory entries which do not contain an entry named a. Note that to ensure this, we must ensure no structural addresses are present in the directory, as this would mean the entry could be in a abstract heap cell. This is used to check that a directory in which a new entry will be created can indeed accept that directory, such as $a[\mathrm{T} \wedge can\_create(b)]$, which describes a directory a, with contents captured in logical variable T that certainly does not contain the entry b. The directory is then a safe place to create the file named A. Notice that $\neg top\_address$ does not prohibit there being addresses deeper inside the directory. They just cannot be directly *in* the directory.

Optional body addresses: Some axioms use *optional* body addresses, where a body address may or may not be present based upon the specific splitting used in a proof. The predicate $maybe(\alpha)$ is used to describe that the address α may or may not exist within a directory tree.

$$maybe(\alpha) \triangleq (\alpha \vee \varnothing)$$

With these predicates, the axioms are largely straightforward translations of the POSIX specification, and match the English versions described earlier. We detail a few, the rest are similar.

`mkdir(path)`: This axiom specifies the directory creation command, and is typical of the file system entry creation command axioms of featherweight POSIXFS. It uses analysis of the `path` expression, breaking it into three components: the *parent* path, which leads up the directory in which the new entry will be added; the *parent directory name*, under which the new entry will be created; and *the new entry name*, which will be added to the parent as an empty directory. This pattern allows the file-system tree resource to be described using an assertion of the form $\alpha^{\mathrm{P}} \mapsto \mathrm{B}[\phi]$. This describes that the directory B in which the new entry will be created is found at path P. The assertion ϕ uses the pattern described above to ensure that no entry

with the target name already exists. Notice that, in the case of $\text{B} = \top$, the path P must be the special *empty* path; this allows creation in the root directory.

`rmdir(path)`: Removing a directory from the file-system tree is much simpler than adding one, as removing empty resource cannot affect the stability of the reasoning. Simple path analysis is performed, showing that the target directory is present at path P. This directory is then removed in the post-condition. Notice that the `rmdir` command can never remove the root directory, as the first path component of PATH cannot be $\varnothing_p$ by the construction of paths.

`link(existing, new)`: This command creates a new file-link entry at path new, linking to the inode of the file at path existing. The pre-condition uses a subtle variation of the common pattern to ensure the new file-link name is not taken. Part of the assertion is $\beta^{\text{P}_2} \mapsto \text{D}[\text{C} \wedge (\mathit{maybe}(\alpha) + \mathit{can\_create}(\text{B}))]$. This describes the directory into which the new entry, named B will be placed. Rather than disallowing *all* body addresses within this directory, the assertion excludes all addresses *except* α, which is used to address the existing source file link. If we did not allow this, the axiom would not allow the creation of the new link inside the directory that contains the source: instances like `link(`$\top$`/a/b, `$\top$`/a/c)`. Such instances are allowed by POSIX, and so much be allowed by us.

`rename(old, new)`: Rename is one of the more complex axioms, due to the number of cases it must consider. This is due to the permutations of resource that `old` and `new` may point to, enumerated in the description of section 6.1.4. When one considers each combination of old and new as directories or files as being a distinct command, each case is actually reasonably simple. The most complex cases are the second and four, where the destination is a directory. In these situations, the source is essentially *deleted*, and then recreated. This incurs the complexity of ensuring there will be no name clash at the `new` path.

`fd := open(path, flags)`: As described in section 6.1.4, has three behaviours governed by `flags`. In all cases, extends the process heap with a new entry for the opened file. In the creation case, creates file system hierarchy entries as well.

`c := read(fd, buff, sz)`: Uses the process heap and file heap to read into the heap buffer. There are three cases, based upon the offset recorded in the file heap, and the length of the file data. A heap buffer of sufficient length must be provided.

`dir := opendir(path)`: The `opendir` command captures the contents of a directory for later enumeration with `readdir`. To ensure that *every* entry is captured, the assertion $\neg \mathit{top\_address}$ is used in the pre-condition. Then, all the names are captured in the logical variable A.

`fn := readdir(dir)`: The non-determinism we permit for directory enumeration is built into the `readdir` axiom. It uses the snapshot of the directory stored in the process heap to non-deterministically select an element known to be in the directory when `opendir` is called. This element is removed from the set in the process heap, and assigned to the return variable.

`t := stat(path)`: There are three `stat` axioms, corresponding to the directory entry, file-link entry and "not found" entry cases. The first two are straight-forward, but the "not found" case requires some subtle analysis. As there is always a root directory in the file system, showing that a path p does not resolve is equivalent to finding a *prefix* of path p that does resolve, and a *postfix* that does not. The axiom contains the analysis:

$$\mathrm{PATH} = \mathrm{P1}/\mathrm{D}/\mathrm{A}/\mathrm{P2} \vee \mathrm{PATH} = \mathrm{P1}/\mathrm{D}/\mathrm{A} \vee ((\mathrm{PATH} = \top/\mathrm{A}/\mathrm{P2} \vee \mathrm{PATH} = \top/\mathrm{A}) \wedge \mathrm{P1} = \varnothing_p \wedge \mathrm{D} = \top)$$

This breaks the path into the prefix $\mathrm{P1}/\mathrm{D}/\mathrm{A}$, and the postfix $\mathrm{P2}$. The abstract heap cells show that the prefix path *does* resolve, but does *not* contain A. Thus, the postfix of the path cannot resolve, and so the overall path cannot resolve.

6.3.4. Admissible and common commands

Of the POSIX file system commands applicable to our fragment, we have given a core subset of 14 of them. Of the others, many can be implemented using our subset. One example is `remove(path)`. The behaviour of `remove` is that, when given a path: if it resolves to a file, invokes `unlink(path)`; if the path resolves to an empty directory, invokes `rmdir(path)`. If the given path does not exist, the command returns -1 to indicate an error. We can easily implement and verify `remove` using our core fragment, as shown in figure 6.3. There are two success cases with the following specifications. We prove that `remove` meets the first of these specifications in figure 6.3.

$$\left\{ \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{ret} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * E \right\} \quad \texttt{ret := remove(path)} \quad \left\{ \texttt{ret} \rightarrow 0 \wedge \alpha^{\mathrm{P}} \mapsto \varnothing * E \right\}$$

$$\left\{ \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{ret} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\varnothing] * E \right\} \quad \texttt{ret := remove(path)} \quad \left\{ \texttt{ret} \rightarrow 0 * \alpha^{\mathrm{P}} \mapsto \varnothing * E \right\}$$

$$\left\{ \quad path(\texttt{path}, \mathrm{P}, \mathrm{B}, \mathrm{A}) \wedge \ \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} \wedge can\_create(\mathrm{A})] * E \quad \right\}$$
$$\texttt{mkdir(path)}$$
$$\left\{ \quad \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} + \mathrm{A}[\varnothing]] * E \quad \right\}$$

$$\left\{ \begin{array}{c} \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \\ \wedge\ \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\varnothing] * E \end{array} \right\}$$
$$\texttt{rmdir(path)}$$
$$\left\{ \quad \alpha^{\mathrm{P}} \mapsto \varnothing * E \quad \right\}$$

$$\left\{ \begin{array}{c} path(\texttt{existing}, \mathrm{P}_1, \mathrm{A}) \wedge path(\texttt{new}, \mathrm{P}_2, \mathrm{D}, \mathrm{B}) \\ \wedge\ \alpha^{\mathrm{P}_1} \mapsto \mathrm{A} : \mathrm{I} * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} \wedge (maybe(\alpha) + can\_create(\mathrm{B}))] * E \end{array} \right\}$$
$$\texttt{link(existing, new)}$$
$$\left\{ \quad \alpha^{\mathrm{P}_1} \mapsto \mathrm{A} : \mathrm{I} * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}] * E \quad \right\}$$

$$\left\{ \quad path(\texttt{path}, \mathrm{P}, \mathrm{A}) \wedge \ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * E \quad \right\}$$
$$\texttt{unlink(path)}$$
$$\left\{ \quad \alpha^{\mathrm{P}} \mapsto \varnothing * E \quad \right\}$$

$$\left\{ \begin{array}{c} path(\texttt{existing}, \mathrm{P}_1, \mathrm{A}) \wedge path(\texttt{new}, \mathrm{P}_2, \mathrm{B}) \\ \wedge\ \alpha^{\mathrm{P}_1} \mapsto \mathrm{A} : \mathrm{I}_1 * \beta^{\mathrm{P}_2} \mapsto \mathrm{B} : \mathrm{I}_2 * E \end{array} \right\}$$
$$\texttt{rename(existing, new)}$$
$$\left\{ \quad \alpha^{\mathrm{P}_1} \mapsto \varnothing * \beta^{\mathrm{P}_2} \mapsto \mathrm{B} : \mathrm{I}_1 * E \quad \right\}$$

$$\left\{ \begin{array}{c} path(\texttt{existing}, \mathrm{P}_1, \mathrm{A}) \wedge path(\texttt{new}, \mathrm{P}_2, \mathrm{D}, \mathrm{B}) \\ * \ \alpha^{\mathrm{P}_1} \mapsto \mathrm{A} : \mathrm{I} * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}\,[\mathrm{C} \wedge (maybe(\alpha) + can\_create(\mathrm{B}))] * E \end{array} \right\}$$
$$\texttt{rename(existing, new)}$$
$$\left\{ \quad \alpha^{\mathrm{P}_1} \mapsto \varnothing * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}] * E \quad \right\}$$

$$\left\{ \quad path(\texttt{existing}, \mathrm{P}_1, \mathrm{A}) \wedge path(\texttt{new}, \mathrm{P}_2, \mathrm{B}) \wedge \ \alpha^{\mathrm{P}_1} \mapsto \mathrm{A}[\mathrm{C} \wedge is\_complete] * \beta^{\mathrm{P}_2} \mapsto \mathrm{B}[\varnothing] * E \quad \right\}$$
$$\texttt{rename(existing, new)}$$
$$\left\{ \quad \alpha^{\mathrm{P}_1} \mapsto \varnothing * \beta^{\mathrm{P}_2} \mapsto \mathrm{B}[\mathrm{C}] * E \quad \right\}$$

$$\left\{ \begin{array}{c} path(\texttt{existing}, \mathrm{P}_1, \mathrm{A}) \wedge path(\texttt{new}, \mathrm{P}_2, \mathrm{D}, \mathrm{B}) \\ \wedge\ \alpha^{\mathrm{P}_1} \mapsto \mathrm{A}[\mathrm{C}_1 \wedge is\_complete] * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}\,[\mathrm{C}_2 \wedge (maybe(\alpha) + can\_create(\mathrm{B}))] * E \end{array} \right\}$$
$$\texttt{rename(existing, new)}$$
$$\left\{ \quad \alpha^{\mathrm{P}_1} \mapsto \varnothing * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C}_2 + \mathrm{B}[\mathrm{C}_1]] * E \quad \right\}$$

$$\left\{ \quad path(\texttt{existing}, \mathrm{P}, \mathrm{A}) \wedge path(\texttt{new}, \mathrm{P}, \mathrm{A}) \wedge \ \alpha^{\mathrm{P}} \mapsto (\mathrm{C} \wedge entry(\mathrm{A})) * E \quad \right\}$$
$$\texttt{rename(existing, new)}$$
$$\left\{ \quad \alpha^{\mathrm{P}} \mapsto \mathrm{C} * E \quad \right\}$$

Figure 6.5.: Axioms for featherweight POSIXFS, continued on the next three pages.

$$\left\{ \quad \texttt{path} \Rightarrow \mathrm{P}/\mathrm{D} \wedge \ \texttt{dir} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{D}[\mathrm{C} \wedge \neg \mathit{top\_address} \wedge \mathit{names}(\mathrm{A})] * E \quad \right\}$$
$$\texttt{dir := opendir(path)}$$
$$\left\{ \quad \exists \mathrm{H}. \ \left(\texttt{dir} \rightarrow \mathrm{H} * \alpha^{\mathrm{P}} \mapsto \mathrm{D}[\mathrm{C} \wedge \mathit{names}(\mathrm{A})] * \mathrm{H} \overset{\mathrm{PH}}{\mapsto} \mathrm{A}\right) * E \quad \right\}$$

$$\left\{ \quad \mathrm{A} \neq \emptyset \wedge \ \texttt{dir} \rightarrow \mathrm{H} \wedge \texttt{fn} \rightarrow \text{-} * \mathrm{H} \overset{\mathrm{PH}}{\mapsto} \mathrm{A} * E \quad \right\}$$
$$\texttt{fn := readdir(dir)}$$
$$\left\{ \quad \mathrm{B} \in \mathrm{A} \wedge \ \texttt{dir} \rightarrow \mathrm{H} * \texttt{fn} \rightarrow \mathrm{B} * \mathrm{H} \overset{\mathrm{PH}}{\mapsto} (\mathrm{A} \setminus \{\mathrm{B}\}) * E \quad \right\}$$

$$\left\{ \quad \texttt{dir} \rightarrow \mathrm{H} * \texttt{fn} \rightarrow \text{-} * \mathrm{H} \overset{\mathrm{PH}}{\mapsto} \emptyset * E \quad \right\}$$
$$\texttt{fn := readdir(dir)}$$
$$\left\{ \quad \texttt{dir} \rightarrow \mathrm{H} * \texttt{fn} \rightarrow \epsilon * \mathrm{H} \overset{\mathrm{PH}}{\mapsto} \emptyset * E \quad \right\}$$

$$\left\{ \quad \texttt{dir} \rightarrow \mathrm{H} * \mathrm{H} \overset{\mathrm{PH}}{\mapsto} \mathrm{A} * E \quad \right\}$$
$$\texttt{closedir(dir)}$$
$$\left\{ \quad \texttt{dir} \rightarrow \mathrm{H} * E \quad \right\}$$

$$\left\{ \begin{array}{c} \mathit{path}(\texttt{path}, \mathrm{P}, \mathrm{B}, \mathrm{A}) \wedge \texttt{flags} \Rightarrow \texttt{O\_CREAT} \\ \wedge \ \texttt{fd} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})] * E \end{array} \right\}$$
$$\texttt{fd := open(path, flags)}$$
$$\left\{ \quad \exists \mathrm{FD}, \mathrm{I}. \ (\texttt{fd} \rightarrow \mathrm{FD} * \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} + \mathrm{A} : \mathrm{I}] * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \varnothing * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, 0) * E) \quad \right\}$$

$$\left\{ \begin{array}{c} \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{flags} \Rightarrow 0 \\ \wedge \ \texttt{fd} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS} * E \end{array} \right\}$$
$$\texttt{fd := open(path, flags)}$$
$$\left\{ \quad \exists \mathrm{FD}. (\texttt{fd} \rightarrow \mathrm{FD} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, 0) * E) \quad \right\}$$

$$\left\{ \begin{array}{c} \mathit{path}(\texttt{path}, \mathrm{P}, \mathrm{A}) \wedge \texttt{flags} \Rightarrow \texttt{O\_TRUNC} \\ \wedge \ \texttt{fd} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS} * E \end{array} \right\}$$
$$\texttt{fd := open(path, flags)}$$
$$\left\{ \quad \exists \mathrm{FD}. (\texttt{fd} \rightarrow \mathrm{FD} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \varnothing * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, 0) * E) \quad \right\}$$

$$\left\{ \begin{array}{c} \texttt{buf} \Rightarrow \mathrm{BUF} \wedge \texttt{sz} \Rightarrow \mathrm{LEN} \wedge \ |\mathrm{BS}_2| = \mathrm{O} \wedge |\mathrm{BS}_3| = |\mathrm{BS}_1| \\ \wedge \ \texttt{fd} \rightarrow \mathrm{FD} * \texttt{c} \rightarrow \text{-} * \mathit{ncells}(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_1) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_2 \cdot \mathrm{BS}_3 \cdot \mathrm{BS}_4 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}) * E \end{array} \right\}$$
$$\texttt{c := write(fd, buf, sz)}$$
$$\left\{ \quad \texttt{fd} \rightarrow \mathrm{FD} * \texttt{c} \rightarrow \mathrm{LEN} * \mathit{ncells}(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_1) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_2 \cdot \mathrm{BS}_1 \cdot \mathrm{BS}_4 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O} + \mathrm{LEN}) * E \quad \right\}$$

Axioms for featherweight POSIXFS (continued from figure 6.5 on the preceding page).

$$\left\{\begin{array}{c}\texttt{buf} \Rightarrow \mathrm{BUF} \wedge \texttt{sz} \Rightarrow \mathrm{LEN} \wedge \ |\mathrm{BS}_2| = \mathrm{O} \wedge |\mathrm{BS}_3| < |\mathrm{BS}_1| \\ \wedge\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \text{-} * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_1) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_2 \cdot \mathrm{BS}_3 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}) * E\end{array}\right\}$$
$$\texttt{c := write(fd, buf, sz)}$$
$$\left\{\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \mathrm{LEN} * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_1) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_2 \cdot \mathrm{BS}_1 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O} + \mathrm{LEN}) * E\ \right\}$$

$$\left\{\begin{array}{c}\texttt{buf} \Rightarrow \mathrm{BUF} \wedge \texttt{sz} \Rightarrow \mathrm{LEN} \wedge \ \mathrm{O} \geq |\mathrm{BS}_2| \\ \wedge\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \text{-} * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_1) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_2 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}) * E\end{array}\right\}$$
$$\texttt{c := write(fd, buf, sz)}$$
$$\left\{\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \mathrm{LEN} * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_1) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} (\mathrm{BS}_2 \cdot \underset{\mathrm{O}-|\mathrm{BS}_2|}{\odot} 0 \cdot \mathrm{BS}_1) * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O} + \mathrm{LEN}) * E\ \right\}$$

$$\left\{\begin{array}{c}\texttt{buf} \Rightarrow \mathrm{BUF} \wedge \texttt{sz} \Rightarrow \mathrm{LEN} \wedge \ |\mathrm{BS}_1| = \mathrm{O} \wedge |\mathrm{BS}_2| = \mathrm{LEN} \\ \wedge\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \text{-} * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{V}) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_1 \cdot \mathrm{BS}_2 \cdot \mathrm{BS}_3 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}) * E\end{array}\right\}$$
$$\texttt{c := read(fd, buf, sz)}$$
$$\left\{\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \mathrm{LEN} * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_2) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_1 \cdot \mathrm{BS}_2 \cdot \mathrm{BS}_3 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O} + \mathrm{LEN}) * E\ \right\}$$

$$\left\{\begin{array}{c}\texttt{buf} \Rightarrow \mathrm{BUF} \wedge \texttt{sz} \Rightarrow \mathrm{LEN} \wedge \ |\mathrm{BS}_1| = \mathrm{O} \wedge |\mathrm{BS}_2| \leq \mathrm{LEN} \\ \wedge\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \text{-} * ncell(\mathrm{BUF}, \mathrm{LEN}, \mathrm{V}) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_1 \cdot \mathrm{BS}_2 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}) * E\end{array}\right\}$$
$$\texttt{c := read(fd, buf, sz)}$$
$$\left\{\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to |\mathrm{BS}_2| * ncell(\mathrm{BUF}, \mathrm{LEN}, \mathrm{BS}_2) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS}_1 \cdot \mathrm{BS}_2 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O} + |\mathrm{BS}_2|) * E\ \right\}$$

$$\left\{\begin{array}{c}\texttt{buf} \Rightarrow \mathrm{BUF} \wedge \texttt{sz} \Rightarrow \mathrm{LEN} \wedge \ \mathrm{O} > |\mathrm{BS}| \\ \wedge\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to \text{-} * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{V}) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}) * E\end{array}\right\}$$
$$\texttt{c := read(fd, buf, sz)}$$
$$\left\{\ \texttt{fd} \to \mathrm{FD} * \texttt{c} \to 0 * ncells(\mathrm{BUF}, \mathrm{LEN}, \mathrm{V}) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{BS} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O})\ \right\}$$

$$\left\{\begin{array}{c}\texttt{to} \Rightarrow \mathrm{O}_1 \wedge \texttt{wh} \Rightarrow \texttt{SEEK\_SET} \wedge \ \mathrm{O}_1 \geq 0 \\ \wedge\ \texttt{fd} \to \mathrm{FD} * \texttt{off} \to \text{-} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}_2) * E\end{array}\right\}$$
$$\texttt{off := lseek(fd, to, wh)}$$
$$\left\{\ \texttt{fd} \to \mathrm{FD} * \texttt{off} \to \mathrm{O}_1 * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}_1) * E\ \right\}$$

Axioms for featherweight POSIXFS (continued from figure 6.5 on page 256).

$$\left\{ \begin{array}{c} \texttt{to} \Rightarrow \mathrm{O}_1 \wedge \texttt{wh} \Rightarrow \texttt{SEEK\_CUR} \wedge \ (\mathrm{O}_1 + \mathrm{O}_2) \geq 0 \\ \wedge\ \texttt{fd} \rightarrow \mathrm{FD} * \texttt{off} \rightarrow \text{-} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}_2) * E \end{array} \right\}$$
$$\texttt{off := lseek(fd, to, wh)}$$
$$\left\{ \ \texttt{fd} \rightarrow \mathrm{FD} * \texttt{off} \rightarrow (\mathrm{O}_1 + \mathrm{O}_2) * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}_1 + \mathrm{O}_2) * E \ \right\}$$

$$\left\{ \begin{array}{c} \texttt{to} \Rightarrow \mathrm{O}_1 \wedge \texttt{wh} \Rightarrow \texttt{SEEK\_END} \wedge \ (|\mathrm{S}| + \mathrm{O}_2) \geq 0 \\ \wedge\ \texttt{fd} \rightarrow \mathrm{FD} * \texttt{off} \rightarrow \text{-} * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{O}_2) * E \end{array} \right\}$$
$$\texttt{off := lseek(fd, to, wh)}$$
$$\left\{ \ \texttt{fd} \rightarrow \mathrm{FD} * \texttt{off} \rightarrow (|\mathrm{S}| + \mathrm{O}_1) * \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, |\mathrm{S}| + \mathrm{O}_1) * E \ \right\}$$

$$\left\{ \ \texttt{fd} \rightarrow \mathrm{FD} * \mathrm{FD} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}, \mathrm{OFFSET}) \ \right\}$$
$$\texttt{close(fd)}$$
$$\left\{ \ \texttt{fd} \rightarrow \mathrm{FD} \ \right\}$$

$$\left\{ \begin{array}{c} \texttt{path} \Rightarrow \mathrm{PATH} \wedge ((\mathrm{PATH} = \mathrm{P/A}) \vee (\mathrm{PATH} = \top \wedge \mathrm{P} = \varnothing_p \wedge \mathrm{A} = \top)) \\ \wedge\ \texttt{t} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta] * E \end{array} \right\}$$
$$\texttt{t := stat(path)}$$
$$\left\{ \ \texttt{t} \rightarrow \mathrm{D} * \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta] * E \ \right\}$$

$$\left\{ \ \texttt{path} \Rightarrow \mathrm{P/A} \wedge \ \texttt{t} \rightarrow \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * E \ \right\}$$
$$\texttt{t := stat(path)}$$
$$\left\{ \ \texttt{t} \rightarrow \mathrm{F} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * E \ \right\}$$

$$\left\{ \begin{array}{c} \texttt{path} \Rightarrow \mathrm{PATH} \\ \wedge\ \mathrm{PATH} = \mathrm{P}_1\mathrm{/D/A/P}_2 \vee \mathrm{PATH} = \mathrm{P}_1\mathrm{/D/A} \vee ((\mathrm{PATH} = \top\mathrm{/A/P}_2 \vee \mathrm{PATH} = \top\mathrm{/A}) \wedge \mathrm{P}_1 = \varnothing_p \wedge \mathrm{D} = \top) \\ \wedge\ \texttt{t} \rightarrow \text{-} * \alpha^{\mathrm{P}_1} \mapsto \mathrm{D}[\mathrm{C} \wedge \neg \mathit{top\_entry}(\mathrm{A}) \wedge \neg \mathit{top\_address}] * E \end{array} \right\}$$
$$\texttt{t := stat(path)}$$
$$\left\{ \ \texttt{t} \rightarrow \mathrm{N} * \alpha^{\mathrm{P}_1} \mapsto \mathrm{D}[\mathrm{C}] * E \ \right\}$$

$$\left\{ \ \texttt{size} \Rightarrow \mathrm{S} \wedge \ \texttt{ptr} \rightarrow \text{-} * E \ \right\}$$
$$\texttt{ptr := malloc(size)}$$
$$\left\{ \ \exists \mathrm{P,V}.\ \texttt{ptr} \rightarrow \mathrm{P} * (\mathrm{P} - 1) \mapsto \mathrm{S} * \mathit{ncells}(\mathrm{P}, \mathrm{S}, \mathrm{V}) * E \ \right\}$$

$$\left\{ \ \texttt{ptr} \Rightarrow \mathrm{P} \wedge \ (\mathrm{P} - 1) \mapsto \mathrm{S} * \mathit{ncells}(\mathrm{P}, \mathrm{S}, \mathrm{V}) * E \ \right\}$$
$$\texttt{free(ptr)}$$
$$\left\{ \ E \ \right\}$$

Axioms for featherweight POSIXFS (continued from figure 6.5 on page 256).

$\{\ \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{ret} \to \text{-} * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * E\ \}$
`ret := remove(path)` $\triangleq$ `local t {`
  `ret := 0`
  `t := stat(path);`
  $\{\ \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{ret} \to 0 * \texttt{t} \to F * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * E\ \}$
  `if t = F`
    $\{\ \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{ret} \to 0 * \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} * E\ \}$
    `unlink(path);`
    $\{\ \texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{ret} \to 0 * \alpha^{\mathrm{P}} \mapsto \varnothing * E\ \}$
  `else if t = D`
    `rmdir(path);`
  `else ret := -1;`
`}`
$\{\ \texttt{ret} \to 0 * \alpha^{\mathrm{P}} \mapsto \varnothing * E\ \}$

Figure 6.3.: The code for `ret := remove(path)`, with the proof sketch for the case when `path` is a file.

We now consider specifying and reasoning about client programs. Noting that `remove` only removes directories if they are empty, we can also specify a command `rmdirRec` that recursively descends from a given directory, removing it and all its descendants. The axiom for this command states that, when provided with the path to a directory, it will remove that entire subtree from the file system:

$$\left\{\ \texttt{path} \Rightarrow \mathrm{P}/\mathrm{N} * \alpha^{\mathrm{P}} \mapsto \mathrm{N}[\mathrm{C} \wedge \mathit{is\_complete}] * E\ \right\}\ \texttt{rmdirRec(path)}\ \left\{\ \alpha^{\mathrm{P}} \mapsto \varnothing * E\ \right\}$$

Another common operation on file systems is the copying of files. Because file sizes can exceed available process memory, they are often copied chunk by chunk. A specification for such an operation is:

$$\left\{\begin{array}{c} \mathit{path}(\texttt{path}, \mathrm{P}_1, \mathrm{A}) \wedge \mathit{path}(\texttt{target}, \mathrm{P}_2, \mathrm{C}, \mathrm{D}) \\ \wedge\ \alpha^{\mathrm{P}_1} \mapsto \mathrm{A} : \mathrm{I}_1 * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})] * \mathrm{I}_1 \overset{\mathrm{F}}{\mapsto} \mathrm{SD} * E \end{array}\right\}$$
$$\texttt{ret := fileCopy(source, target)}$$
$$\left\{\ \exists\, \mathrm{I}_2.\alpha^{\mathrm{P}_1} \mapsto \mathrm{A} : \mathrm{I} * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{A} : \mathrm{I}_2] * \ \mathrm{I}_1 \overset{\mathrm{F}}{\mapsto} \mathrm{SD} * \mathrm{I}_2 \overset{\mathrm{F}}{\mapsto} \mathrm{SD} * E\ \right\}$$

The pre-condition states that a file exists at the path given by `source`, and that the file we will create at `target` does not yet exist. The disjunction on the value of TGT handles the case where we may be creating a file in the root directory. The post-condition ensures that the `target` file has been created, and a corresponding entry I_2 added to the file heap

with identical contents to that of the `source` file. The implementation code and associated proof is in the appendix, section A.1.5.

6.4. Software installer example

One type of program that interacts extensively with the file system is a *software installer*. Modern software is typically downloaded by the users in an archive, and is thus initially unusable. The goal of a software installer is to take this archive and place the contents at the correct points in the user's file system, such that the software can run. Most installers also check that the users computer meets certain requirements (e.g. no conflicting software is already installed), remove previous versions of the software, and either remove or alter incompatible configuration files.

Good software installers either successfully complete installation, or make no changes to the machine. Here, we develop an good installer for the fictional software "Widget Version 2". It supersedes "Widget Version 1", but is unfortunately incompatible with any V1 configuration files that users may have created. Widget V2 consists of a program executable, '$widgProg$' and a data file, '$widgData$'. We follow common conventions for placing these files in the users file system [64]. The two Widget files will be placed in the directory '$\top/opt/widget/$'. The program will be made usable by creating a hard link from '$\top/usr/bin/widget$' (the system directory for user provided executables) to the file '$\top/opt/widget/widgProg$'.

The installer must contend with the possible existence of Widget V1, as well as unrelated files that the user may have chosen to store at the paths it wants to use. For example, a user may have created a directory at '$\top/usr/bin/widget$' - our installer should not remove this, as that could damage other components of their system. However, we *will* remove it if it is just a file, as we assume it to be a Widget V1 executable. Similarly, we must remove '$\top/opt/widget$' only if it is a directory. Finally, if any user has placed a Widget V1 configuration file in their home directory, e.g. '$\top/home/user_n/.wconf$', we choose to remove that as well.

Considering these requirements, the Widget V2 installer must:

① Check to see if entries already exist at the locations we wish to place Widget V2 files, and abort installation if they are not Widget V1 entries.

② Remove Widget V1 entries, if they were found.

③ Check for a Widget V1 configuration file in each users home directory, removing it if found.

④ Copy Widget V2 files to the target location on the file system.

⑤ Make a link to the Widget V2 executable, so the user can run it.

The installation of a simple, two file program has become quite a complex task. We therefore specify our intuitions about good behaviour and prove that our installer matches them. We build a pre-condition for our installer out of several sub-assertions. In these, the assertion $\otimes_{\mathrm{X}\in\mathrm{E}}\phi$ is the iterated version of $+$, interpreted as $\phi_1 + \cdots + \phi_{|\mathrm{E}|}$ where each ϕ_i has X bound to a distinct member of E. We also extend the logical environment to include pairs of logical values, instrumented file systems and process heaps, and extend the heap assertions to include such logical variables. In this way, we can write logical variables that capture invariants about the entire heap.

$$
\begin{array}{rcl}
v2Dir_P & \triangleq & \mathrm{widgProg} : \mathrm{J} + \mathrm{widgData} : \mathrm{K} \\
v2Files_P & \triangleq & \mathrm{J} \overset{\mathrm{F}}{\mapsto} \mathrm{PROG} * \mathrm{K} \overset{\mathrm{F}}{\mapsto} \mathrm{DAT} \\
home_P & \triangleq & \mathrm{home}\left[\otimes_{(\mathrm{N},\mathrm{C})\in\mathrm{HOMES}}\mathrm{N}\left[\begin{array}{c}(\mathrm{C} \wedge \neg top\_entry(.\mathrm{wconf})) \\ + \ (\varnothing \vee \exists \mathrm{I}.\ .\mathrm{wconf} : \mathrm{I}) \\ \wedge\ \neg top\_address\end{array}\right]\right] \\
bin_P & \triangleq & \mathrm{bin}\left[\begin{array}{c}(\mathrm{B} \wedge \neg top\_entry(\mathrm{widget})) + \\ (\varnothing \vee entry(\mathrm{widget})) \wedge \neg top\_address\end{array}\right] \\
opt_P & \triangleq & \mathrm{opt}\left[\begin{array}{c}(\mathrm{T} + (\varnothing \vee \mathrm{widget}[\mathrm{T}_w \wedge \neg\exists\alpha.\ \Diamond\alpha]) \\ \wedge\neg\ top\_address\end{array}\right]
\end{array}
$$

Each of these fragments describes the states that parts of the file system may be in for the Widget installer to run safely. The directory entries that make up the Widget version 2 program are described by $v2Dir_P$; their corresponding file data by $v2Files_P$. We require that these files be in some unspecified place in the file system, with this location is stored in variable `instLoc`.

The $home_P$ resource captures all the home directories on the system, along with the fact that some of them will contain a '$.wconf$' file containing Widget V1 configuration data for that user. We use a logical variable, HOMES to store the pair of all the home directory names and their contents. This allows us to construct a loop invariant as we iterate across each directory looking for the '$.wconf$' file.

The bin_P resources captures the normal UNIX executables directory, that may or may not contain a "widget" entry. Notice we require that there be no structural addresses present in any widget directory that may exist, as any such directory will be deleted. Also, we require for most of our components that there be no top addresses. This allows

us to assert the absence of a file (the $\neg top\_entry(\cdots)$ assertions) safe in the knowledge that it is not present in another frame. The use of $top\_address$ does not stop there being holes deeper within the structures captured by the logical variables, so we are still minimal in the resource we demand.

We combine these descriptions into a pre-condition for the program. The pre-condition makes extensive use of structural addresses, localising the proof to just the parts of the file system touched by the installer. It also snapshots the initial state of resource in a logical variable W, to show that nothing changes in the event of failure.

$$P_{ins} \triangleq \begin{array}{l} \texttt{instLoc} \to \text{IL} * \text{W} \wedge \delta^{\text{IL}} \mapsto v2Dir_P * v2Files_P * \\ \alpha^{\top} \mapsto home_P * \gamma^{\top/usr} \mapsto bin_P * \beta^{\top} \mapsto opt_P \end{array}$$

If the installer fails, we expect that the file system should be unchanged. If it does not fail, we expect that the program should be successfully installed. There should be no other outcomes. We describe a successful installation with the following predicates:

$$\begin{array}{rcl} v2Dir_Q & \triangleq & \text{widgProg} : \text{J}' + \text{widgDat} : \text{K}' \\ v2Files_Q & \triangleq & \text{J}' \overset{\text{F}}{\mapsto} \text{PROG} * \text{K}' \overset{\text{F}}{\mapsto} \text{DAT} \\ home_Q & \triangleq & \text{home} \left[\otimes_{(\text{N},\text{C}) \in \text{HOMES}} \text{N} \left[\begin{array}{c} \text{C} \\ \wedge \neg top\_entry(\text{`}.wconf\text{'}) \\ \wedge \neg top\_address \end{array} \right] \right] \\ bin_Q & \triangleq & \text{bin}[\text{B} + \text{widget} : \text{J}'] \\ opt_Q & \triangleq & \text{opt}[\text{T} + \text{widget}[v2Dir_Q]] \end{array}$$

The program post-condition is built from these predicates.

$$Q_{ins} \triangleq \begin{array}{l} \exists \text{RET}, \text{J}', \text{K}'.\ \texttt{ret} \to \text{RET} * \texttt{instLoc} \to \text{IL} * \\ \neg \text{RET} \Rightarrow \text{W} \wedge \\ \text{RET} \Rightarrow \left(\begin{array}{c} \delta^{\text{IL}} \mapsto v2Dir_P * \alpha^{\top} \mapsto home_Q \\ * \ \beta^{\top} \mapsto opt_Q * \ \gamma^{\top/usr} \mapsto bin_Q \\ * \ v2FilesQ * \textbf{true} \end{array} \right) \end{array}$$

Installation can fail if entries already exist at the paths we wish to use (point ① above). In this case, the program assigns variable `ret` to **false** and we use the W to show that the file system is unchanged. If it is true, i.e. the installation was successful, then file system reflects this: the '$\top/opt/widget$' directory exists and is suitably populated by the new

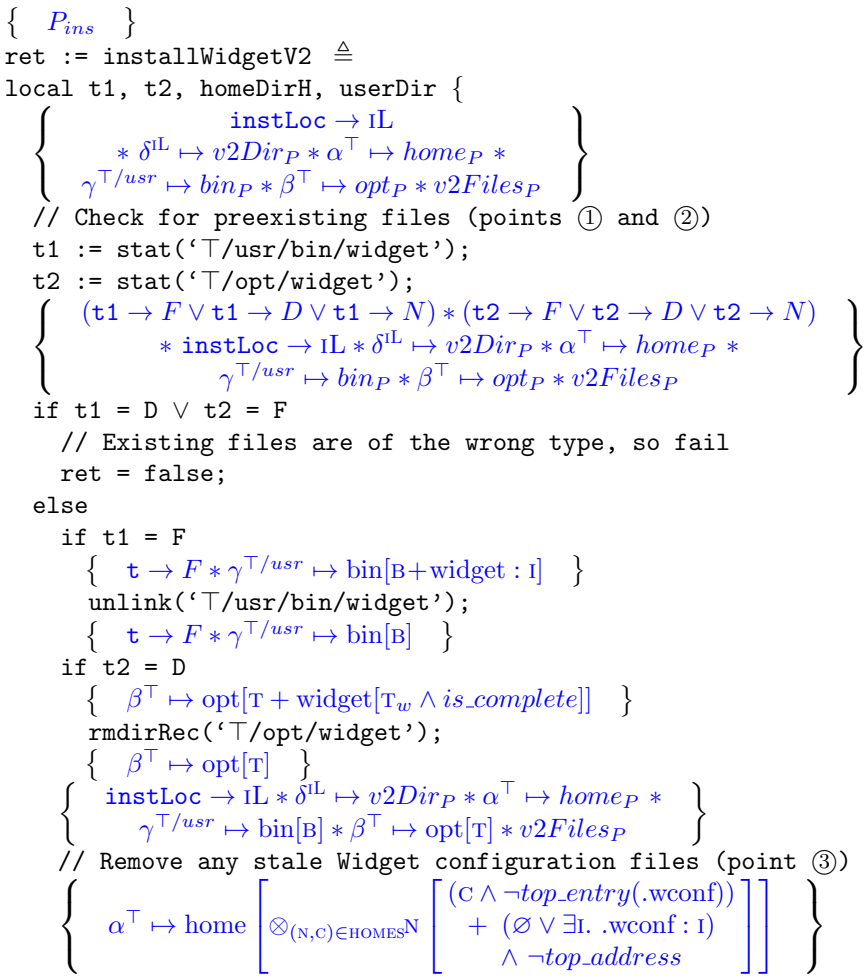


Figure 6.4.: Code and proof for Widget V2 software installer (continued on next page)

Widget V2 files, no user specific configuration files exist, and the '$\top/usr/bin/widget$' executable points to the newly installed program. Notice that any Widget V1 files are subsumed by the predicate **true**, awaiting garbage collection, and that the logical variables $\text{T}, \text{B}, \cdots$ are witness to the fact that no other resource in our pre-condition has changed.

The code for the installer, and proof that it meets this specification $\{P_{ins}\}$ `installWidgetV2` $\{Q_{ins}\}$, is given in figure 6.4. As another testament to the compositional properties of our reasoning, notice that we reuse the `rmdirRec`, to delete existing directory subtrees, and `fileCopy` to copy the binary and data files from the installer directory to the target location. Throughout the sketch we implicitly use the frame rule and semantic consequence to localise the resource we are acting on.

Figure 6.4.: Widget V2 Software Installer program (continued)

6.5. Summary

This chapter has introduced a structural separation logic specification for the *POSIX* file system library. This forms the second of our two large case studies, demonstrating the entire process of analysing and formalising a "real world" library using our techniques. Our key contributions are:

1. **Featherweight POSIXFS**: Section 6.1 introduced a novel subset of POSIX, identifying and describing a set of core commands used in file systems. This subset is both accurate with respect to the POSIX specification, and amiable to formal analysis.
2. **Axiomatic specification for POSIX file systems**: Using structural separation logic, section 6.3.3 gives a axiomatic specification of our POSIX subset. The axioms are accurate to POSIX[4], providing an unambiguous description of the behaviour of the commands.
3. **Demonstration of naturally stable promises**: Our specifications are given as local axioms by using path promises. This demonstrates that the promise technique scales to a real example.
4. **Local reasoning for POSIX file systems programs**: The installer example of section 6.4 demonstrates that our specification can be used to reason about non-trivial file system manipulating programs.

[4]Expect for the two cases noted, `readdir` and `malloc`.

7. Obligations

The naturally stable promises of chapter 5 allow passive access to information in the surrounding context. However, they restrict the set of commands that can be considered to those which can never break promises. This prevents some useful libraries from being specified, as seen with the list library in section 5.3.2. There, when using promises to state the list index from which an abstract heap cell was abstractly allocated, the `remove` had to be restricted to the *last* element of a list in order to preserve stability.

This chapter introduces *obligations*, which act as the dual of promises. Whereas the promises associated with an abstract heap cell give information about the shape of the context, obligations associated with a cell restrict the shape of the data within the cell. Example obligations might be "This abstract heap cell always contains root tree node N", or "The number of list elements in this abstract heap cell is at least 3".

When using obligations, the set of command axioms is unrestricted, subject to the standard requirement of atomic soundness 12. Proving commands atomically sound means showing that the obligations can never be violated. Thus, ann `appendChild` command acting on a cell with the obligation "Always contains root tree node N" would have to ensure that, after the command, the cell still contained root node N. A list `remove` command acting on a cell with the obiligation "number of list elements is at least 3" could never reduce the number of cells to less than 3. Obligations thus ensure that promises are kept. Whenever a non-naturally stable promise is issued during abstract allocation, an associated obligation is created to ensure the promise is kept. When the promise is destroyed, so is the obligation.

Our work on obligations is less mature than other chapters of these thesis. Here, we develop the initial theory of obligations, demonstrating that they restore full function to the list `remove` command. We show how to reasoning using obligations. We will reexamine our DOM specification (chapter 4) in the light of this work, and demonstrate how the technique could be used to improve the specification, by enabling new behaviours or by tightening the footprints for commands. At the end of this chapter, we speculate on how obligations may allow us to reason about *symbolic links*. Symbolic links are a type of file that *redirect* file system path resolution to continue at another point in the file-system

tree.

Related work

As discussed in section 5, promises can be seen as both a style of rely-guarantee, deny-guarantee reasoning, and of passivity system. This is similarly true of obligations. Just as a promise associated with a heap cell is something about the environment the cell can rely on, an obligation associated with a heap cell is a *guarantee* to the environment that the given invariant will always hold. As promises and obligations can be split (by splitting the abstract heap cells they are attached to), the work is more similar to Deny-Guarantee than Rely-Guarantee. However, both systems are typically used to enable concurrent update to the *same* resource, with the guarantees and rely (denys) being used to ensure a safe protocol for this update.

Thus, we again prefer to link this work to *permission systems*. Separation logic, traditionally using the Boyland-style fractional permissions [10], means that each heap cell in a footprint can be in two states: write and read, or just read. Promises and obligations offer similar statements, but over *invariants*. As discussed in 5, a promise associated with a heap cell is a "read-only" view of some data. It can be seen as a permission $\pi < 1$ on that data. However, oblilgations are mroe expressive. Rather than the cell that issued the promise also being associated with $\pi < 1$, the obligation states only that it must maintain some invariant. Consider a heap cell $\mathrm{x} \mapsto \mathrm{v}$. Using normal permissions, this splits into $\mathrm{x} \overset{0.5}{\mapsto} \mathrm{v} * \mathrm{x} \overset{0.5}{\mapsto} \mathrm{v}$, rendering both passive. However, if v is *structured* data using promises and obligations, it can be split $\exists \alpha. \mathrm{x}_{\alpha:I} \mapsto \mathrm{v}_1 * \alpha^{\Cap:I} \mapsto \mathrm{v}_2$. Here, not only is the data is splt, but some invariant I is shared. Thus, x can still be updated, as long as I is still true.

7.1. Lists access via indexing

Recall the list indexing promises of section 5.3.2. These promises are naturally stable only if the `remove` command can only remove the final list element. This prevents any issued indexing promises from being broken. Here, we work with the unrestricted `remove` command, and prevent promises being broken with with *obligations*.

We update the two index promises with their associated obligations. Again, one states the index at which an abstract heap cell can be found, whilst the other states the length of the data in a cell. Informally, the promise and obligation pairs are:

$\alpha \Leftarrow i$ If α maps to $\Cap$, this denotes a promise that the data comes from index i in the list. If α does not map to $\Cap$, this denotes *an obligation* to α to keep the body address $\mathbf{x}$ at index i of the list.

$\alpha \Rightarrow n$ If α maps to $\Cap$, this denotes an obligation to keep the length of the data to exactly n. If α does not map to $\Cap$, this denotes a promise that the data contained within the cell at α will be of length n.

Using the first as an obligation, we can split the element 4 from the list $\mathrm{L} \mapsto [3 \otimes 1 \otimes 4 \otimes 2$ in such a way that the cell L still knows that the element must be at index 3. As , we write promises as superscripts on heap addresses, we write obligations as subscripts:

$$\mathrm{L}_{\alpha \Leftarrow 3} \mapsto [3 \otimes 1 \otimes 4 \otimes \alpha] * \alpha^{\Cap \Leftarrow 3} \mapsto 2$$

By associating this obligation with L, we know we must never alter the data within that cell in any way that would make the body address α no longer the 3rd child. This, by itself, would severely restrict what can be done with the α cell. One could never safely abstractly allocate part of the $3 \otimes 1 \otimes 4$ sub-list, as the resultant L addressed cell would not have enough information to guarantee that α is at index 3. This problem can be overcome with the *length promise and obligation*. To allow, say, $3 \otimes 1$ to be allocated at β, we issue it with a *length obligation*, which grants L a *length promise* from β:

$$\mathrm{L}^{\beta \Rightarrow 2}_{\alpha : \Leftarrow 3} \mapsto [\beta \otimes 4 \otimes \alpha] * \alpha^{\Cap \Leftarrow 3} \mapsto 2 * \beta_{\Cap \Rightarrow 2} \mapsto 3 \otimes 1$$

This promise states that the data compressed into the β address of α will always contain exactly two elements. The promise is given the β name, so that when β is compressed, it can be removed. The promise will be paired with whichever cell has the β body address, allowing it to be transfered as the tree is further cut up. For example, if the cell α is split via abstract allocation. Now, considered in isolation, the assertion $\mathrm{L}^{\beta \Rightarrow 2}_{\alpha \Leftarrow 3} \mapsto [\beta \otimes 4 \otimes \alpha]$ is obliged to always keep α at index 3. It can ensure that this is true because the β cell has promised it will contain data of length 2.

7.2. Obligations

Obligations are the *dual* of promises. Whereas promises tell a datum about the shape of context which fits around it, obligations are restrictions on the datum that ensure it always conforms to a specific shape. When data is associated with an unstable promise then, somewhere, a matching *obligation* exists that ensures the promise will always hold.

Figure 7.1 extends a promise-carrying datum into a *promise- and obligation-carrying datum*. Before, in figure 5.5, this datum had three promises: one head promise, and two body promises. Now, it still has these promises, but has gained a *head obligation* and two *body obligations*.

There is strong symmetry here: promises received from super-data are *head promises*, consisting of *head data* (definition 91) with exactly one address (⋒). Promises received from sub-data are *body promises*, made up of *body data* (definition 92) containing no holes. Obligations reverse this relationship: obligations *to* super-data (*head obligations*) consist of *body data*. Obligations made to sub-data (*body obligations*) are made up of head data.

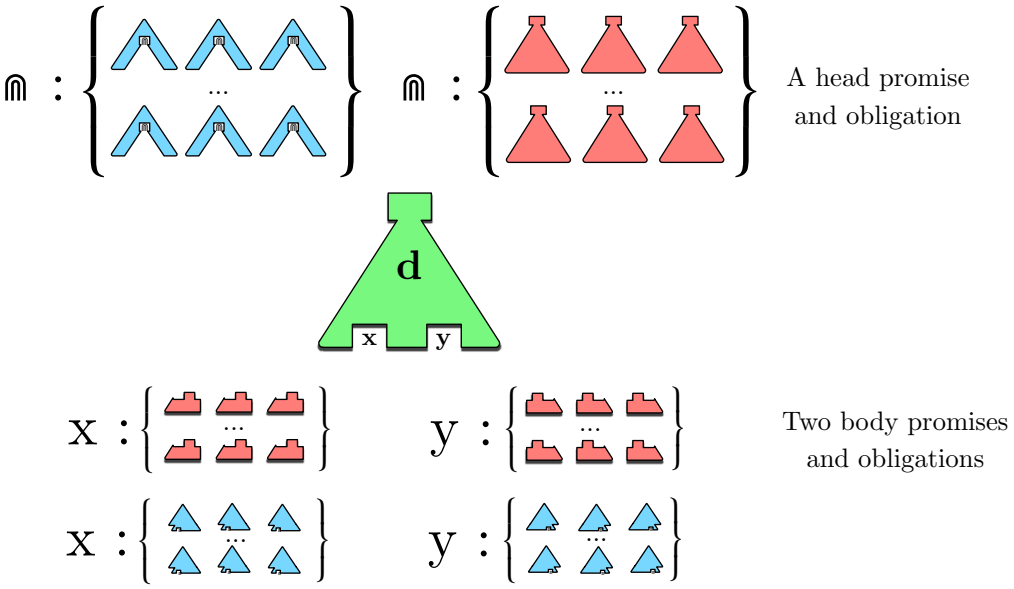


Figure 7.1.: A datum with three promises, and three obligations

.

Any data carrying obligations must *enforce* them, so that the combination of the data and promises is sufficient to show that the obligation is satisfied. It is not legal to construct data which does not satisfy its obligations. This implies that a datum must have extracted sufficient promises from any sub- and super-data to ensure the obligations remain true.

Compression using naturally stable promises could verify that the promises held by comparing the closure of the data against the promises. This was sufficient, as naturally stable promises can never be broken; as long as the data currently satisfies the promise,

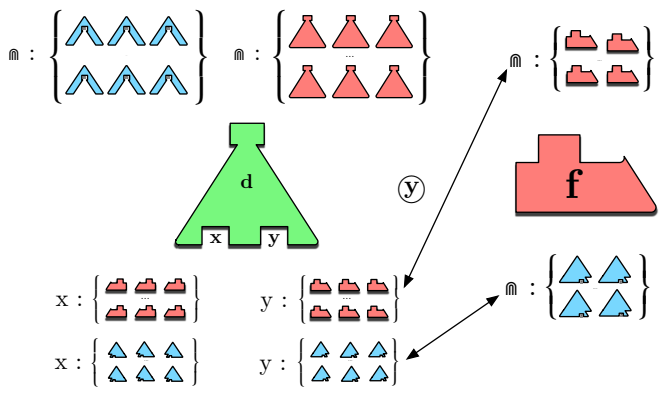


Figure 7.2.: Compression when using obligations

the stability of the reasoning ensures that it must *always* have satisfied it. With unstable promises, we instead check promises against the obligations. As the obligations bound the range of changes that are possible, if the promises are satisfied by the obligations, they must always have been true.

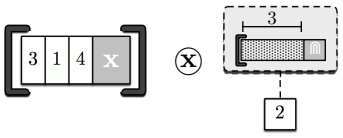
Index and length promises for lists

As an example of promise- and obligation-carrying data, consider figure 7.3. This considers the promises and obligations for list indexing and lengths for the list library with an `item` command. It shows the combinations of promises and obligations that allow or disallow compression in a variety of cases.

7.2.1. Formalising obligations

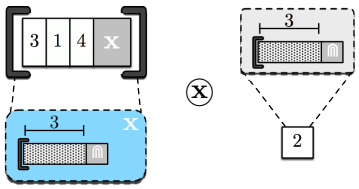
We now formalise the intuitions of obligations introduced in the last section. As stated, obligations are, in many ways, symmetric to promises. The notions of head and body data (definitions 91 and 92 respectively) are retained unchanged. However, rather than head obligations used head data, they will use *body data*. This is because, rather than stating what information will be provided *by* the super-data, they state what information is provided *to* to the super-data.

Parameter 22 (Head obligations)**.** Assume a set of **head obligations** HeadObls, ranged over by ω^H, with the type:



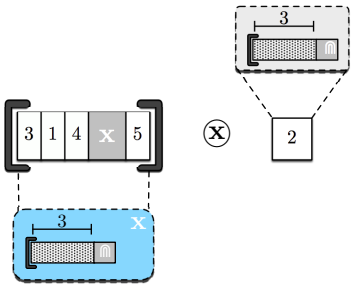
✘

Whilst the super-data provides exactly 3 left neighbours, it is not *obliged* to, and so may break the promise. Thus, compression is not allowed.



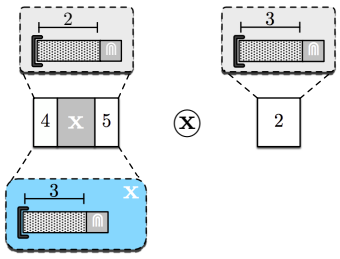
✓

Compression is allowed as super-data provides exactly 3 left neighbours, and is obliged to always do so until the **x** cell is collapsed.



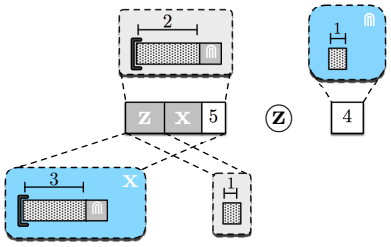
✓

Compression is allowed as the super-data provides exactly 3 left neighbours, and is obliged to always be so until the **x** cell is collapsed. Further cells to the right of the **x** body address do not break the obligation.



✓

Compression is allowed as the super-data is obliged to keep **x** at index 3. It ensures this via a combination of its own promise to be at index 2, and the local data stored within it.



✓

Here, the **x** data is not present. However, we are still obliged to ensure that the body address **x** it is at index 3. The data containing body address **x** has this obligation, which it meets via a promise from its parent to keep it at index 2, and a promise from **z** to be of length 1. The **z** data is therefore obliged to be of length 1. Therefore, compression is allowed.

Figure 7.3.: Examples of promises for list indexing. The promise  indices that structural address ⋒ is at index 3 of list l. The associated obligation to **x** is  . The promise that the data within **z** is of length 1 is  . The associated obligation to the super-data is  .

$$\textsc{HeadObls} \subseteq \{\Cap\} \times \textsc{BodyData}$$

Similarly, body obligations make guarantees *to* the sub-data that will eventually compress into a body address. These obligations describe the shape of heads they will compress into. Ergo, body obligations use *head data*.

Parameter 23 (Body obligations)**.** Assume a set of **body obligations** BodyObls, ranged over by ω^B, with the type:

$$\textsc{BodyObls} \subseteq \textsc{StructAddrs} \times \textsc{HeadData}$$

The set of obligations is then all groups of body obligations that may include a head obligation. This is exactly as was defined for promises but in reverse.

Definition 137 (Obligations)**.** Recall that HeadData $\cup$ BodyData is ranged over by $\overline{\underline{d}}, \overline{\underline{d}}_1, \cdots, \overline{\underline{d}}_n$. Then, given a set of head and body obligations HeadObls and BodyObls (parameters 22 and 23), the set of **obligation sets** (or just **obligations**), ranged over by $\Omega, \Omega_1, \cdots, \Omega_n$, is defined as:

$$\textsc{Obligations} = \left\{ \Omega \in \mathcal{P}\left(\begin{array}{c}\textsc{HeadObls}\\ \cup \\ \textsc{BodyObls}\end{array}\right) \;\middle|\; \begin{array}{l}\nexists \mathbf{x} \in \textsc{StructAddrs}.\\ \exists \overline{\underline{d}}_1, \overline{\underline{d}}_2.\ \{(\mathbf{x}, \overline{\underline{d}}_1), (\mathbf{x}, \overline{\underline{d}}_2)\} \subseteq \Omega\end{array} \right\}$$

Let $\Omega \in \textsc{Obligations}$. We write the removal of the obligation for any address $\mathbf{x} \in \textsc{StructAddrs}$ from Ω as:

$$\Omega - \mathbf{x} \triangleq \{(\mathbf{y}, \overline{\underline{d}}) \mid (\mathbf{y}, \Omega) \in \Pi, \mathbf{y} \neq \mathbf{x}\}$$

With obligations, we can extend promise-carrying data to be *promise- and obligation-carrying data*. As with promise-carrying data, each obligation must refer to either the head, or a body address. More strongly, the data and promises in the group must conspire to satisfy the obligations. It is, by construction, impossible to construction obligation-carrying data that does not satisfy its obligations.

Definition 138 (Promise- and obligation-carrying data)**.** The set of **promise- and obligation-carrying data** PROMOBSDATA, ranged over by $\mathbf{pod}, \mathbf{pod}_1, \cdots, \mathbf{pod}_n$ is defined as:

$$\textsc{PromObsData} \subseteq \textsc{Data}_{\Cap} \times \textsc{Promises} \times \textsc{Obligations}$$

where every promise or obligation is either associated with the head, or with a body address in the data. That is, for all $(d, \Pi, \Omega) \in \textsc{PromObsData}$:

$$\forall (\mathbf{x}, \overline{\underline{d}}) \in \Pi \cup \Omega.\ \mathbf{x} = \Cap \text{ or } \mathbf{x} \in \text{addrs}(d)$$

Moreover, all obligations must be fulfilled: for all $\omega \in \Omega$

1. If this is a head obligation, then the body closure (definition 93) of this data satisfies the obligation: $\omega = (\Cap, \underline{d})$ implies $\downarrow (d, \Pi) \subseteq \underline{d}$.
2. If this is a body obligation to $\mathbf{x}$, then the closure of the data with respect to $\mathbf{x}$ (definition 96) satisfies the obligation: $\omega = (\mathbf{x}, \overline{d})$ implies $\circlearrowleft_{\mathbf{x}} ((d, \Pi)) \subseteq \overline{d}$.

We now defined compression, which no longer checks that the promises hold with respect to the data, instead checking with respect to the *obligations*. This ensures that, no matter what shape the data had been, as long as it was obliged to hold the shapes expected, the compression can proceed. There is no need to check that the obligations hold, as obligation-carrying data is always valid. In section A.3, we prove that obligation-carrying data is a valid structural addressing algebra. The proof is similar to the equivalent proof in chapter 5.

Definition 139 (Compression for obligation-carrying data)**.** The **compression function** for promise- and obligation-carrying data, $\text{comp}_O : \textsc{StructAddrs} \to \textsc{PromObsData} \to \textsc{PromObsData} \rightharpoonup \textsc{PromObsData}$ is defined as: for all $\mathbf{pod}_1 = (d_1, \Pi_1, \Omega_1)$ and $\mathbf{pod}_2 = (d_2, \Pi_2, \Omega_2)$

$$\text{comp}_O(\mathbf{x}, \mathbf{pod}_1, \mathbf{pod}_2) = \begin{pmatrix} \text{comp}_{\Cap}(\mathbf{x}, d_1, d_2), \\ (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap), \\ (\Omega_1 - \mathbf{x})) \cup (\Omega_2 - \Cap)) \end{pmatrix}$$

where:

1. The underlying data compression is defined: $\text{comp}_{\Cap}(\mathbf{x}, d_1, d_2)$ defined.

2. If it exists, the head promise made to $\mathbf{pd}_2$ by $\mathbf{pd}_1$ is satisfied by a body obligation on $\mathbf{pd}_2$: $(\Mapsto, \overline{d}_2) \in \Pi_2 \implies \exists(\mathbf{x}, \overline{d}_1) \in \Omega_1.\ \overline{d}_1 \subseteq \overline{d}_2$.
3. If it exists, the body promise made to $\mathbf{pd}_1$ by $\mathbf{pd}_2$ is satisfied by the head obligation on $\mathbf{pd}_2$: $(\mathbf{x}, \underline{d}_1) \in \Pi_1 \implies \exists(\Mapsto, \underline{d}_2) \in \Omega_2.\ \underline{d}_1 \subseteq \underline{d}_2$.

An instance $\mathrm{comp}_O(\mathbf{x}, \mathbf{pod}_1, \mathbf{pod}_2)$ is written $\mathbf{pod}_1 \circledast \mathbf{pod}_2$, overloading the compression notation.

7.3. Reasoning with obligations

We now consider reasoning with obligations, applying the technique to the list indexing problem. We also revisit the DOM and file-systems example of chapters 4 and 6.

7.3.1. Obligations for list indexing

We now turn to axiomatising the `item` command for the lists library. We first formalise the promises and obligations we will use.

Definition 140 (Promises and obligations for index lists)**.** The **list index head data with respect to x** and the **list length body data with respect to x** are defined as:

$$\begin{aligned} \Leftarrow i &= \{\overline{d} \mid d \in \overline{d}, d = [pl_1 \otimes \mathbf{x} \otimes pl_2], |pl_1| = i\} \\ \Rightarrow i &= \{\underline{d} \mid d \in \underline{d}, |d| = i\} \end{aligned}$$

The **promises for the list library** are defined as:

$$\textsc{Promises} = \begin{array}{ll} & \bigcup_{i \in \mathbb{N}} \{(\Mapsto, \Leftarrow i)\} \quad \text{List index promises} \\ \cup & \bigcup_{i \in \mathbb{N}, \mathbf{x} \in \textsc{StructAddrs}} \{(\mathbf{x}, \Rightarrow i\} \quad \text{List length promises} \end{array}$$

The **obligations for the list library** are defined as:

$$\textsc{Obligations} = \begin{array}{ll} & \bigcup_{i \in \mathbb{N}, \mathbf{x} \in \textsc{StructAddrs}} \mathbf{x} \Leftarrow i \quad \text{List index obligations} \\ \cup & \bigcup_{i \in \mathbb{N}} \Mapsto \Rightarrow i \quad \text{Partial list length obligations} \end{array}$$

Here, we are using more than just head promises. Many promises and obligations may accrue as abstract allocation occurs. To ease notation, we define logical expressions for denoting sets of promises and obligations.

Definition 141 (Logical expressions for list promises and obligations)**.** The logical expressions for the list library with item command and associated interpretation function are extended with the following expressions:

$$
\begin{array}{ll}
\langle\![\emptyset]\!\rangle(\Gamma) & \triangleq \emptyset \\
\langle\![\mathrm{A} \Leftarrow \mathrm{E}]\!\rangle(\Gamma) & \triangleq \begin{cases} \Gamma(\mathrm{A}) \Leftarrow \langle\![\mathrm{E}]\!\rangle(\Gamma) & \text{if } \Gamma(\mathrm{A}) \in \textsc{StructAddrs}, \langle\![\mathrm{E}]\!\rangle(\Gamma) \in \mathbb{N} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\langle\![\mathrm{A} \Rightarrow \mathrm{E}]\!\rangle(\Gamma) & \triangleq \begin{cases} \Gamma(\mathrm{A}) \Rightarrow \langle\![\mathrm{E}]\!\rangle(\Gamma) & \text{if } \Gamma(\mathrm{A}) \in \textsc{StructAddrs}, \langle\![\mathrm{E}]\!\rangle(\Gamma) \in \mathbb{N} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\langle\![\mathrm{E}_1, \cdots, \mathrm{E}_n]\!\rangle(\Gamma) & \triangleq \{\langle\![\mathrm{E}_1]\!\rangle(\Gamma), \cdots, \langle\![\mathrm{E}_n]\!\rangle(\Gamma)\}
\end{array}
$$

We then define a promise- and obligation carrying abstract list heap cell assertion.

Definition 142 (Promise- and obligation carrying heap assertion)**.** The abstract heap assertions of the list library are extended to promise- and obligation-carrying data with:

$$
\langle\!|\mathrm{A}^{\mathrm{E}_P}_{\mathrm{E}_O} \mapsto \phi|\!\rangle^{\Gamma} = \left\{ \mathbf{a} \mapsto (d, \Pi, \Omega) \;\middle|\; \begin{array}{l} \mathbf{a} = \Gamma(\mathrm{A}), d \in \langle\!|\phi|\!\rangle^{\Gamma}, \\ \Pi = \langle\![\mathrm{E}_P]\!\rangle(\Gamma), \\ \Omega = \langle\![\mathrm{E}_O]\!\rangle(\Gamma) \end{array} \right\}
$$

The assertion $\mathrm{A}^{\emptyset}_{\emptyset} \mapsto \phi$ is written $\mathrm{A} \mapsto \phi$. Similarly, $\mathrm{A}^{\emptyset}_{O} \mapsto \phi$ is written $\mathrm{A}_O \mapsto \phi$ and $\mathrm{A}^{P}_{\emptyset} \mapsto \phi$ is written $\mathrm{A}^P \mapsto \phi$.

We can replicate the set of equivalences given in section 5.3.2 using obligations. Notice that in all cases, the L cell retains enough information in each case to ensure the obligation.

$$\mathrm{L} \mapsto [1 \otimes 2 \otimes 3] \quad \Rrightarrow_\Gamma \quad \exists \alpha.(\mathrm{L} \mapsto [1 \otimes \alpha \otimes 3] * \alpha \mapsto 2$$
$$\mathrm{L} \mapsto [1 \otimes 2 \otimes 3] \quad \Rrightarrow_\Gamma \quad \exists \alpha.(\mathrm{L}_{\alpha \Leftarrow 1} \mapsto [1 \otimes \alpha \otimes 3] * \alpha^{\Cap \Leftarrow 1} \mapsto 2)$$

$$\exists \alpha.(\mathrm{L}_{\alpha \Leftarrow 1} \mapsto [1 \otimes \alpha \otimes 3] * \alpha^{\Cap \Leftarrow 1} \mapsto 2)$$
$$\Rrightarrow_\Gamma$$
$$\exists \alpha, \beta.(\mathrm{L}^{\beta \Rightarrow 1}_{\alpha \Leftarrow 1} \mapsto [\beta \otimes \alpha \otimes 3] * \alpha^{\Cap \Leftarrow 1} \mapsto 2 * \beta_{\Cap \Rightarrow 1} \mapsto 1)$$

$$\exists \alpha, \beta.(\mathrm{L}^{\beta \Rightarrow 1, \alpha \Rightarrow 1}_{\alpha \Leftarrow 1} \mapsto [\beta \otimes \alpha \otimes 3] * \alpha^{\Cap \Leftarrow 1}_{\Cap \Rightarrow 1} \mapsto 2 * \beta_{\Cap \Rightarrow 1} \mapsto 1)$$
$$\Rrightarrow_\Gamma$$
$$\exists \alpha, \beta, \gamma.(\mathrm{L}^{\beta \Rightarrow 1, \alpha \Rightarrow 1}_{\alpha \Leftarrow 1, \gamma \Leftarrow 2} \mapsto [\beta \otimes \alpha \otimes \gamma] * \alpha^{\Cap \Leftarrow 1}_{\Cap \Rightarrow 1} \mapsto 2 * \beta_{\Cap \Rightarrow 1} \mapsto 1 * \gamma^{\Cap \Leftarrow 2} \mapsto 3)$$

The axiom for the list indexing command is essentially unchanged from that given with promises. Here, it carries an arbitrary set of obligations, as the command is passive, changing no data and so invalidating nothing.

$$\{\alpha^{\Cap \Leftarrow \mathrm{I}}_{\mathrm{O}} \mapsto \mathrm{J} * \mathtt{j} \to \text{-} * E \wedge e \Rightarrow \mathrm{I} \wedge |\mathrm{J}| = 1\}$$
$$\mathtt{j := item}(e)$$
$$\{\alpha^{\Cap \Leftarrow \mathrm{I}}_{\mathrm{O}} \mapsto \mathrm{J} * \mathtt{j} \to \mathrm{J} * E\}$$

The key contribution of obligations is enabling commands that *could* invalidate promises. The problematic command was the `remove`, which in its most general form could break the item index promise. Here, we can restore the full behavioour by ensuring `remove` have an *empty* set of obligations. It can thus never invalidate a promise.

$$\left\{\alpha^{\emptyset}_{\emptyset} \mapsto \mathrm{I} * E \wedge e \Rightarrow \mathrm{I}\right\}$$
$$\mathtt{remove}(e)$$
$$\left\{\alpha^{\emptyset}_{\emptyset} \mapsto \varnothing * E\right\}$$

With this obligation, `remove` is atomically sound.

Lemma 25 (The `remove` command is atomically sound)**.** *The* `remove` *command is atomically sound, satisfying 12 of the framework.*

Proof. Recall the `remove(`e`)` action given in definition 28:

$$[\![\texttt{remove}(e)]\!](s) \triangleq \begin{cases} \{s[\mathrm{L} \mapsto [pl_1 \otimes pl_2]]\} & \text{if } s(\mathrm{L}) = [pl_1 \otimes ([\![e]\!])(s) \otimes pl_2] \\ \{\lightning\} & \text{otherwise} \end{cases}$$

The pre-condition of `remove(`e`)` is $\alpha^{\emptyset}_{\emptyset} \mapsto \mathrm{I} * E \wedge e \Rightarrow \mathrm{I}$. After interpretation, the provides views of the form $\mathbf{x} \mapsto (i, \emptyset, \emptyset) * varHeap$, where $varHeap$ is the arbitrary variable resource produced by E. The lack of promises and obligations means we have no information about the frame, but do know that mutations cannot destabilise the reasoning. After notating that the post-condition also has no promises and obligations, atomic soundness of the command then follows via a similar argument to that of lemma 10. □

More interesting is what would happen if we picked a bad specification for `remove`: not having obligations, using an non-empty set of obligations $\alpha^{\emptyset}_{O} \mapsto \cdots$, and changing the obligations in the axiom all result in unsoundness.

1. Without obligations, changes to data within the α cell could invalidate any promise regarding list lengths, as there would be no way to know what promises had been issued depending on the data.
2. If we included arbitrary obligations, so the post-condition contained $\alpha^{\emptyset}_{O} \mapsto \varnothing$, it would *not* abstract the result of the command, as such a cell represents invalid data, and so a false post-condition.
3. If we had removed the obligations, so the post-condition contained $\alpha^{\emptyset}_{\emptyset} \mapsto \varnothing$, then we could propose a frame of the form $\beta^{P} \mapsto \alpha$ where P is the promise associated with the obligation that α previously had. This frame would be invalidated by the update, and so the command would not be atomically sound.

7.3.2. Enhancing the specifications of DOM

Obligations allow us to further tighten DOM axiomatisation we gave in chapter 4. There are two axioms in the DOM library of 4 that arguably involve more data access than is restrictly necessary. The first, `item`, acts much like the `item` command discussed with lists. We can thus use list index promises and obligations to reduce the footprint. The second, `appendChild` uses

7.3.3. The `item` command footprint

Recall the item command of DOM, `j := f.item(e)` which extracts the node at index e from the forest identified by `f`. We have already considered a virtually identical situation

example in section 7.1, albeit on a single list. DOM has many forests rather than a single list, but it is straightforward to extend the promises and obligations to describe *which* forest is under consideration. We can extend the list index head data from $\Uparrow \Leftarrow \mathrm{I}$, which describes the single list with the head address at index I, to $\mathrm{F} \Leftarrow \mathrm{I}$, which describes describes all DOM trees where the $\Uparrow$ address is the I^{th} child of node with forest identifier F. When used as a head promise, provides exactly what is needed for the `item` command. Using promises and obligations of this form, the `f.item(i)` command can be axiomatised as:

$$\left\{ \texttt{f} \rightarrow \mathrm{FID} * \texttt{n} \rightarrow \texttt{-} * \alpha_{\mathrm{O}}^{\mathrm{FID} \Leftarrow \mathrm{I}} \mapsto \mathrm{S_N}[\gamma]_{\mathrm{FS}} * E \wedge e \Rightarrow \mathrm{I} \right\}$$
$$\texttt{n := f.item(e)}$$
$$\left\{ \texttt{f} \rightarrow \mathrm{FID} * \texttt{n} \rightarrow \mathrm{N} * \alpha_{\mathrm{O}}^{\mathrm{FID} \Leftarrow \mathrm{I}} \mapsto \mathrm{S_N}[\gamma]_{\mathrm{FS}} * E \right\}$$

The footprint has been reduced to only the node that will be returned. As well as more concisely statement the effect of the command, it enables more concurrency with respect to the elements of the forest FID, as each node can be managed in a different thread.

7.3.4. The `appendChild` command footprint

One example that is not based on the simple examples already presented in this chapter is the `p.appendChild(c)` command. This command, which moves the node identified by `c` to be the last child of the node identified by P, currently has the axiom:

$$\{\texttt{p} \rightarrow \mathrm{P} * \texttt{c} \rightarrow \mathrm{C} * \texttt{o} \rightarrow \texttt{-} * \alpha \mapsto \mathrm{S_P}[\gamma]_{\mathrm{FS}_1} * \beta \mapsto \mathrm{S'_C}[\mathrm{T} \wedge \mathit{is\_complete}]_{\mathrm{FS}_2} \wedge \mathrm{S} \neq \#\mathrm{text}\}$$
$$\texttt{o := n.appendChild(m)}$$
$$\{\texttt{p} \rightarrow \mathrm{P} * \texttt{c} \rightarrow \mathrm{C} * \texttt{o} \rightarrow \mathrm{C} * \alpha \mapsto \mathrm{S_P}[\gamma \otimes \mathrm{S'_C}[\mathrm{T}]_{\mathrm{FS}_2}]_{\mathrm{FS}_1} * \beta \mapsto (\varnothing_f \vee \varnothing_g)\}$$

Notice that the `c` node must have no body addresses. This states that the sub-tree under `c` is complete, which is needed to ensure that the node `p` is not a descendent of `c`. Unfortunately, it also gives `appendChild` a larger than ideal footprint, ensuing that no concurrent analysis of the nodes under `c` is possible.

We can correct this excessive footprint with a "No descendent with identifier n" promise. An abstract heap making this promise states that it *never* contains a node with identifier n, anywhere in the sub-data. This promise is not naturally stable, as the `appendChild` command can break it by placing a node with the disqualified identifier into the subtree. The other axioms cannot break the obligation, and can just carry it unchanged.

Definition 143 (No such node promise & obligation)**.** The body data **no such node** is defined as:

$$\neg n = \{d \in \text{DOMDATA} \mid \nexists \mathbf{x}, d_1, d_2.\ d = d_1 \circledast n[d_2]\}$$

With this, the body promises and head obligations are defined as:

$$\begin{aligned} \text{BODYPROMS} &= \{(\mathbf{x}, \neg n) \mid \mathbf{x} \in \text{STRUCTADDRS}, n \in \text{NODEIDS}\} \\ \text{HEADOBLS} &= \{(\Cap, \neg n) \mid n \in \text{NODEIDS}\} \end{aligned}$$

There are no body obligations or head promises.

The "no such node" body data $\neg n$ describes all trees which do not contain node identifier n. The allowable promises thus describe a context containing *any* tree data without a specific node. As this is not stable, the associated obligation states that a given cell never contains the same node.

Using this promise/obligation pair, we can create an axiom where the children of `c` are described with a body address and promise stating that, whatever the children are, they do not contain the target parent node. We can then axiomatise the children of the `c` node using body address, retaining on the fact that `p` is not one of them. This node is moved to be a child of `p`, which by the obligation cannot created a cycle. We must check, however, that the obligations associated with the abstract cell containing `p` will allow a child `c`, to ensure we are not breaking another obligation that may be in force. The `appendChild` axiom becomes:

$$\left\{\texttt{p} \to \text{P} * \texttt{c} \to \text{C} * \texttt{o} \to \text{-} * \alpha^{\text{P1}}_{\text{O1}} \mapsto \text{S}_{\text{P}}[\gamma]_{\text{FS1}} * \beta^{\text{P2},\delta:\neg\text{P}}_{\text{O2}} \mapsto \text{S'}_{\text{C}}[\delta]_{\text{FS2}} \wedge \text{S} \neq \#\text{text} \wedge (\neg\text{C}) \notin \text{O}_1\right\}$$

```
o := n.appendChild(m)
```

$$\left\{\texttt{p} \to \text{P} * \texttt{c} \to \text{C} * \texttt{o} \to \text{C} * \alpha^{\text{P1},\delta:\neg\text{P}}_{\text{O1}} \mapsto \text{S}_{\text{P}}[\gamma \otimes \text{S'}_{\text{C}}[\delta]_{\text{FS2}}]_{\text{FS1}} * \beta^{\text{P2}}_{\text{O2}} \mapsto (\varnothing_f \vee \varnothing_g)\right\}$$

As before, soundness of `appendChild` depends upon the sub-tree under node `m` *not* containing the target parent node `m`. We assure this via the final conjunct of the pre-condition, $(\neg\text{C}) \notin \text{O}_1$. This states that the obligations associated with α permit a descendent node with identifier C. This ensures the stability of the system, as `appendChild` is the only axiom which can add a node to a sub-tree. All other commands need only carry the promises and obligations untouched.

Notice that this axiom is *weaker* than our original, promise free version. Here, we are us-

ing only head obligations; this implies that any abstract cell containing no body addresses must have *no* obligations. Our original axiom mandated $\beta \mapsto \mathrm{S'}_{\mathrm{M}}[\mathrm{T} \wedge \mathit{is\_complete}]_{\mathrm{FS}_2}$ has no body addresses, beneath the node with identifier M, and so has no obligations, trivially satisfying $(\neg \mathrm{C}) \notin \mathrm{O}_1$. This weaker axiom allows more flexible use of the data. The following parallel use of `appendChild` and `nodeName` is only possible with the obligations system, in which we elide the program variable resource to avoid clutter:

$$\{\alpha \mapsto \mathrm{S_P}[\varnothing]_{\mathrm{PFS}} \otimes \mathrm{T_C}[\mathrm{U_D}[\varnothing_f]_{\mathrm{DFS}}]_{\mathrm{CFS}}\}$$

$$\left\{\exists \beta, \gamma, \delta, \epsilon, \zeta. \begin{array}{c} \alpha \mapsto \beta \otimes \gamma * \beta \mapsto \mathrm{S_P}[\delta]_{\mathrm{PFS}} * \gamma^{\epsilon:\neg \mathrm{P}} \mapsto \mathrm{T_C}[\epsilon]_{\mathrm{CFS}} \\ * \ \delta \mapsto \varnothing_f * \epsilon^{@:\neg \mathrm{P}}_{\zeta:\neg \mathrm{P}} \mapsto \mathrm{U_D}[\zeta]_{\mathrm{DFS}} * \zeta^{@:\neg \mathrm{P}} \mapsto \varnothing_f \end{array}\right\}$$

$$\begin{array}{c} \{\beta \mapsto \mathrm{S_P}[\delta]_{\mathrm{PFS}} * \gamma^{\epsilon:\neg \mathrm{P}} \mapsto \mathrm{T_C}[\epsilon]_{\mathrm{CFS}}\} \\ \texttt{p.appendChild(c)} \\ \{\beta^{\epsilon:\neg \mathrm{P}} \mapsto \mathrm{S_P}[\delta \otimes \mathrm{T_C}[\epsilon]_{\mathrm{CFS}}]_{\mathrm{PFS}} * \gamma \mapsto \varnothing_f\} \end{array} \quad \Big\| \quad \begin{array}{c} \left\{\epsilon^{@:\neg \mathrm{P}}_{\zeta:\neg \mathrm{P}} \mapsto \mathrm{U_D}[\zeta]_{\mathrm{DFS}}\right\} \\ \texttt{s := d.nodeName} \\ \left\{\epsilon^{@:\neg \mathrm{P}}_{\zeta:\neg \mathrm{P}} \mapsto \mathrm{U_D}[\zeta]_{\mathrm{DFS}}\right\} \end{array}$$

$$\left\{\exists \beta, \gamma, \delta, \epsilon, \zeta. \begin{array}{c} \alpha \mapsto \beta \otimes \gamma * \beta^{\epsilon:\neg \mathrm{P}} \mapsto \mathrm{S_P}[\delta \otimes \mathrm{T_C}[\epsilon]_{\mathrm{CFS}}]_{\mathrm{PFS}} * \gamma \mapsto \varnothing \\ * \ \delta \mapsto \varnothing_f * \epsilon^{@:\neg \mathrm{P}}_{\zeta:\neg \mathrm{P}} \mapsto \mathrm{U_D}[\zeta]_{\mathrm{DFS}} * \zeta^{@:\neg \mathrm{P}} \mapsto \varnothing_f \end{array}\right\}$$

$$\{\alpha \mapsto \mathrm{S_P}[\otimes \mathrm{T_C}[\mathrm{U_D}[\varnothing_f]_{\mathrm{DFS}}]_{\mathrm{CFS}} \varnothing]_{\mathrm{PFS}}\}$$

7.3.5. Symbolic links

We now briefly outline *symbolic links* for file systems (chapter 6, and speculate on how obligations may be able to reason about these. Whereas hard links point directly to the inode of an entry, a *symbolic link* contains the *path* to an entry. Consider figure 7.4. The directory in the bottom right contains a *symbolic link file*. This symbolic link file references the directory at path $\top/\mathit{Music}/\mathit{Classical}$. When resolving paths that go "through" $\top/\mathit{Documents}/\mathit{Music}/\mathit{Classical}/$, the resolution process will *follow* the symbolic link, and continue at the target of it. Therefore, the path $\top/\mathit{Documents}/\mathit{Music}/\mathit{Classical}/\mathit{adagio.mp3}$ resolves to the file with inode 6.

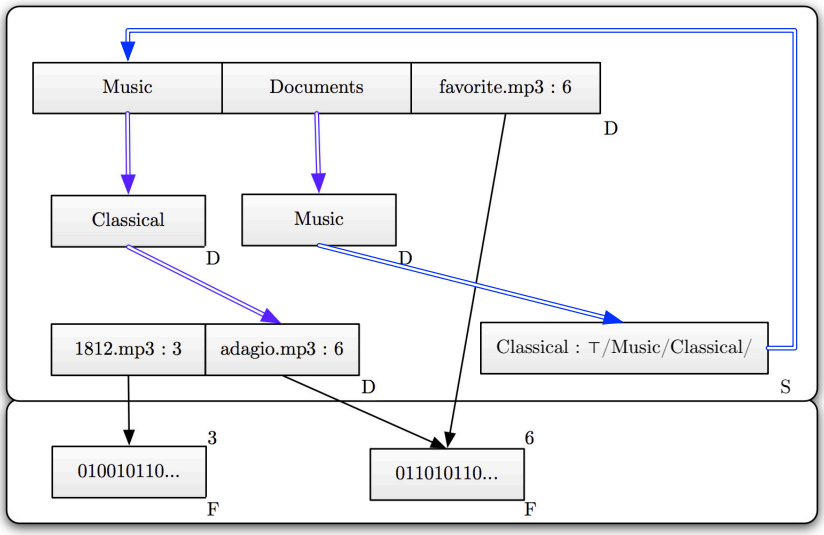


Figure 7.4.: A POSIX file-system tree where the directory in the bottom right contains a symbolic link. The hard links traversed by resolution are highlighted.

There are two major obstacles to reasoning with symbolic links:

1. The addition of symbolic links means that paths need not follow the inductive structure of the file-system tree. Resolving a path may involve traversing symbolic links, and resolving each symbolic link can involve resolving *further* symbolic links, ad-infinitum. POSIX avoids such unbounded path resolution by providing a constant integer `SYMLOOP_MAX`. After `SYMLOOP_MAX` symbolic links have been encountered whilst resolving a path, resolution fails with an error.
2. Symbolic links also break the intuition that the data used to identify sub-data is *disjoint* from the sub-data itself. With symbolic links, the data to be updated can be involved in the resolution process. For example, a descendent of a directory d could be a symbolic link pointing to d itself. By identifying d via a path using that symbolic link, the contents of d are directly used in identifying d. If we wanted to, for example, remove the contents of d using `rmdirRec(⊤/path/to/d/through/symlink)`, at some point during the removal the symbolic link will be deleted, and the path will stop resolving! Such a situation is not possible with the linear paths we are currently using.

Nonetheless, promises and obligations provide some of the machinery needed to management symbolic links. Linear paths were described using simple head promises, which

provide a sub-datum d with the shape of super-data c that resolves the path. Any symbolic links that only use resource in c can be handled using an enriched notion of our resolve function acting on c. As symbolic links will use sub-data from c that is not a direct ancestor of d (that is, the symbolic links), obligations would be required to ensure the stability of the system.

However, the second obstacle remains difficult. Our framework is based upon *splitting* data. Even promises and obligations split the data, into local data which can be updated, and that which is promised from the frame. To solve this problem, we would seem to need a system that simultaneously allows the *local* view *and* a *global* view of the complete data. Our framework, with promises and obligations, provides all the needed parts to build this global view, but currently has no elegant method for analysing it. We will continue to pursue this problem.

7.4. Summary

This chapter has introduced *obligations*, a system for restricting the possible updates to abstract heap cells. Obligations, when combined with promises, combine to give a general theory of *passive access* for structured data, where data can be shared among many abstract heap cells, and updates restricted to ensure this sharing is safe. This allows:

1. **Richer smaller axioms for libraries**: Obligations allow a greater range of commands to be specified using promises, by removing the restriction that promises be naturally stable. We have demonstrated this with the list `item` and `remove` commands, which are not possible with either structural separation logic or promises alone.
2. **Smaller axioms for DOM**: We can provide even smaller axioms for the DOM library, allowing a greater separation of DOM tree data. This will enable a much richer notion of concurrent DOM in future work.
3. **Beginnings of permission system for structured data**: This system is an initial attempt at a *permission system* for structured data. Promises provide one part of this, offering passive read access to data not directly *owned* by a thread. Obligations provide the other. Rather than just use "read" or "read and write" permissions, obligations allow a richer range of updates by stating "Any change is permitted, up to the given obligation invariants".

As stated, the obligations system is less mature than naturally stable promises and structural separation logic. One obvious issue is the verbose nature of the bookkeeping, requiring significant effort during proof construction. We will continue to develop the obligations system in future work.

8. Conclusions

This thesis has primarily been concerned with techniques for, and examples of, *library reasoning*. By developing reasoning that takes an *abstraction first* rather than *implementation first* approach to specifying libraries, we have been able to give novel axiomatic semantics for a range of libraries. We have covered simple data structures (such as lists and trees), updated pre-existing axiomatic specifications for real libraries (with DOM), and introduced entirely new specifications for complex libraries which are not immediately amenable to local reasoning (such as POSIX). By allowing a pragmatic mix of standard heap models and our abstractions, we have enabled easier verification of programs that *use* these libraries. This thesis has touched on many aspects of program verification and has, inevitably, opened many new questions that remain unanswered.

Our primary technical contribution has been *structural separation logic*, a program logic for local reasoning with mixed *structured* and *unstructured* data. We have provided a general presentation of structural separation logic, showing how the technique can be applied to many different *libraries* that use a mix of highly structured data and normal heap manipulation. When verifying client programs that use libraries, this mix of high level abstraction with low level heap manipulation allows easier program verification than would be possible with separation logic alone. We have demonstrated this both with our DOM reasoning example in section 4.4, and more extensively with our featherweight POSIX case study.

By design, the abstract heap cells of our logic inherit the separation afforded by the separating conjunction. Beyond the simple flat heaps used in our examples, this means we can interface easily with the developments of the wider separation logic community. One particular area we wish to interface with is *tooling*. We can represent abstract heap cells within existing separation logic tools where, to the tools proof engine, they are harmless passengers in the assertions and are simply removed by the frame rule. We can then extend the proof engines to be *aware* of the abstract cells, including their creation and destruction via abstract allocation, and update via axioms. We have already begun experiments with this, embedding a simple version of structural separation logic into the Verifast tool [45] with promising results. However, the correct long-term solution to representing our

abstract data remains unclear. We will aggressively pursue automation in future work, focusing on experimentation and proving real examples.

We have shown how structural separation logic can give an axiomatic specification to DOM that has smaller footprints than was previously possible. Using structural separation logic for DOM allows easy composition of DOM reasoning with language reasoning. This will allow reasoning about *web applications*, which are written in JavaScript, interacting with the browsers page representation via the DOM library. There is now significant work on local program reasoning with JavaScript using variants on the standard separation logic heap model [34]. Structural separation logic allows this to be combined with our DOM axioms to obtain a system capable of reasoning about web applications. An initial version of this work appeared in [66], and we hope to work with Smith et al. in extending this to a fuller approach.

As with previous DOM axiomatic semantics, our axioms are justified against a high-level operational semantics. This can be unconvincing, as we author the semantics specifically to allow verification of the axioms. To obtain a stronger soundness result, it will be necessary to link our approach an underlying *implementation*. Structural separation logic is well placed to achieve this, as we can mix heap implementation code with our abstraction. We are currently working with Raad on refinement for the DOM library. We have developed an implementation in C, and by using reasoning techniques for refinement being developed by her and Wheelhouse, we can show that this soundly refines our axiomatic specifications. This enables a verified *concurrent* DOM implementation, the first of its kind.

Local reasoning is not always easily applicable to libraries. Our axioms for POSIX show that even well-designed libraries do not always localise their data access. Even in these situations, we believe that the compositionality provided via locality can be a useful concept. As such, we have introduced *promises and obligations* to act act as a *permission* system for abstract data. They enable data it to be split into *read-only* global data and *read-write* local data, and so allow locality where it would otherwise be impossible. Promises and obligations are only the first step in understanding richer notions of separation and composition on abstract structured data. Our inability to easily specify symbolic links that go *through* abstractly allocated data indicate that it may be helpful to have two simultaneous views of data; one "cut-up" via abstract allocation, and one "complete", using promises and obligations, to give context to the localised sub-data.

However, the current form has been sufficient to prove interesting programs that use the POSIX file system. Our approach is one of the first POSIX specifications focused entirely on verifying *client programs*, rather than POSIX implementations. Although our subset is already useful, the POSIX standard is large. Gzik is working on extending featherweight

POSIXFS to handle more of the POSIX file-system specification, including additional commands, richer symbolic link and non-linear path support, and (with us) *errors*. We have so far ignored errors without our system, so that any command not correctly used is treated as a faulting case. POSIX typically reports such command invocations as non-fatal *errors*, allowing programs to recover and proceed in another manner. It will be interesting to see the challenges this poses for reasoning.

This thesis is but a single step in local reasoning for client programs that use libraries. As users of libraries vastly outnumber authors, such a focus will help enable more software achieve correctness through verification. We will continue to push this approach in our research, analysing and axiomatising more libraries. We will continue to extend the theory to handle the realities of what we find, rather than work only on idealised libraries designed with verification in mind.

A. Appendices

In these appendices, we include both some proofs too straightforward for, or similar to, proofs from the main text, and additional program proofs using structural separation logic.

A.1. Program proofs

This section contains various programs and their associated proofs, demonstrating further uses of structural separation logic.

A.1.1. Proof for `contains`

$$\left\{ \quad \mathtt{k} \to \mathrm{K} * \mathtt{b} \to \text{-} * \mathrm{L} \mapsto [\mathrm{A}] \quad \right\}$$

```
b := contains(k) ≜ local i {
 i := getFirst();
```

$$\left\{ \quad (\mathrm{A} = \varnothing \wedge \mathtt{i} \to 0) \vee (\exists \mathrm{I},\mathrm{R}.\ \mathrm{A} = \mathrm{I} \otimes \mathrm{R} * \mathtt{i} \to \mathrm{I}) * \mathrm{L} \mapsto [\mathrm{A}] \quad \right\}$$

```
 b := false;
```

$$\left\{ \begin{array}{l} \exists \mathrm{I},\mathrm{B},\mathrm{S},\mathrm{R}.\ \mathtt{k} \to \mathrm{K} * \mathtt{b} \to \mathrm{B} * \mathrm{L} \mapsto [\mathrm{A}] * \mathtt{i} \to \mathrm{I} \wedge \\ (\mathrm{A} = \mathrm{S} \otimes \mathrm{I} \otimes \mathrm{R} \wedge \mathrm{I} \neq 0) \vee (\mathrm{A} = \mathrm{S} \wedge \mathrm{I} = 0) \wedge \\ ((\mathrm{K} \in \mathrm{S} \wedge \mathrm{B} = \mathbf{true}) \vee (\mathrm{K} \notin \mathrm{S} \wedge \mathrm{B} = \mathbf{false})) \end{array} \right\}$$

```
 while (i ≠ 0)
```

$$\left\{ \quad \mathtt{i} \to \mathrm{I} * \mathtt{k} \to \mathrm{K} * \mathtt{b} \to \mathrm{B} * \exists \mathrm{J}, \beta.\ (\alpha \mapsto (\mathrm{I} \otimes \mathrm{J} \otimes \beta) \vee \alpha \mapsto [\beta \otimes \mathrm{I}]) \quad \right\}$$

```
  if (i = k)
   b := true;
  else
   skip;
  i := getRight(i)
}
```

$$\left\{ \quad \mathtt{k} \to \mathrm{K} * ((\mathrm{K} \in \mathrm{A} \wedge \mathtt{b} \to \textsc{true}) \vee (\mathrm{K} \notin \mathrm{A} \wedge \mathtt{b} \to \textsc{false})) * \mathrm{L} \mapsto [\mathrm{A}] \quad \right\}$$

A.1.2. Proof of `l := n.stringLength`

$$\left\{\ \mathtt{n} \to \mathrm{N} * \mathtt{l} \to \text{-} * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S} \wedge \mathit{is\_complete}]_{\mathrm{FS}}\ \right\}$$

```
l := n.stringLength ≜ local r {
 l := 0;
 r := n.substringData(l, 1);
```

$$\left\{ \begin{array}{c} \exists \mathrm{S1},\mathrm{S2},\mathrm{R},\mathrm{L}.\ \mathtt{n} \to \mathrm{N} * \mathtt{l} \to \mathrm{L} * \mathtt{r} \to \mathrm{R} * \\ \left(\begin{array}{c} (\mathrm{R} = \mathbf{null} \wedge \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S} \wedge \mathrm{S1}]_{\mathrm{FS}}) \\ \vee \\ (\mathrm{R} \neq\ \wedge \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S} \wedge \mathrm{S1} \cdot \mathrm{R} \cdot \mathrm{S2}]_{\mathrm{FS}}) \end{array} \right) \wedge \mathrm{L} = |\mathrm{S1}| \end{array} \right\}$$

```
 while (r != null)
  l := l + 1;
  r := n.substringData(l, 1);
}
```

$$\left\{\ \exists \mathrm{L}.\ \mathtt{n} \to \mathrm{N} * \mathtt{l} \to \mathrm{L} * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{FS}} \wedge \mathrm{L} = |\mathrm{S}|\ \right\}$$

A.1.3. Proof of `s := n.value` in the text node case

$$\left\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \right\}$$

```
s := n.value ≜ local l, name {
 name := n.nodeName;
```

$$\left\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \mathtt{l} \to \text{-} * \mathtt{name} \to \#\mathrm{text} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \right\}$$

```
 if name = "#text"
```

$$\left\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \mathtt{l} \to \text{-} * \mathtt{name} \to \#\mathrm{text} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \right\}$$

```
  l := n.stringLength;
  s := n.substringData(0, l)
```

$$\left\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \mathrm{VAL} * \mathtt{l} \to |\mathrm{VAL}| * \mathtt{name} \to \#\mathrm{text} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \right\}$$

```
 else
```

$$\left\{\ \mathbf{false}\ \right\}$$

```
  s := null
}
```

$$\left\{\ \mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}} \wedge \mathtt{s} \to \mathrm{VAL}\ \right\}$$

$\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \}$
`s := n.value` $\triangleq$ `local l, name {`
`name := n.nodeName;`
$\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \mathtt{l} \to \text{-} * \mathtt{name} \to \#\mathrm{text} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \}$
`if name = "#text"`
$\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \text{-} * \mathtt{l} \to \text{-} * \mathtt{name} \to \#\mathrm{text} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \}$
`l := n.stringLength;`
`s := n.substringData(0, l)`
$\{\ \mathtt{n} \to \mathrm{N} * \mathtt{s} \to \mathrm{VAL} * \mathtt{l} \to |\mathrm{VAL}| * \mathtt{name} \to \#\mathrm{text} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}}\ \}$
`else`
$\{\ \mathbf{false}\ \}$
`s := null`
`}`
$\{\ \mathtt{n} \to \mathrm{N} * \alpha \mapsto \#\mathrm{text}_{\mathrm{ID}}[\mathrm{VAL}]_{\mathrm{FS}} \wedge \mathtt{s} \to \mathrm{VAL}\ \}$

Figure A.1.: Proof of `s := n.value` in the text node case.

A.1.4. Proof of `v := n.childValue`

$\{\ \mathtt{n} \to \mathrm{N} * \mathtt{v} \to \text{-} * \alpha \mapsto \mathrm{NM}_{\mathrm{ID}}[\#\mathrm{text}_{\mathrm{TID}}[\mathrm{VAL}]_{\mathrm{FS1}} \otimes \beta]_{\mathrm{FS2}}\ \}$
`v := n.childValue` $\triangleq$ `local cs, c {`
`cs := n.childNodes;`
$\{\ \exists \mathrm{FS3}, \mathrm{FID}.\ \mathtt{n} \to \mathrm{N} * \mathtt{v} \to \text{-} * \mathtt{c} \to \text{-} * \mathtt{cs} \to \mathrm{FID} * \alpha \mapsto \mathrm{NM}_{\mathrm{ID}}[\#\mathrm{text}_{\mathrm{TID}}[\mathrm{VAL}]_{\mathrm{FS1}} \otimes \beta]_{\mathrm{FS3}} \wedge \mathrm{FID} \in \mathrm{FS3} \wedge \mathrm{FS2} \subseteq \mathrm{FS3}\ \}$
`c := cs.item(0);`
$\{\ \exists \mathrm{FS3}, \mathrm{FID}.\ \mathtt{n} \to \mathrm{N} * \mathtt{v} \to \text{-} * \mathtt{c} \to \mathrm{TID} * \mathtt{cs} \to \mathrm{FID} * \alpha \mapsto \mathrm{NM}_{\mathrm{ID}}[\#\mathrm{text}_{\mathrm{TID}}[\mathrm{VAL}]_{\mathrm{FS1}} \otimes \beta]_{\mathrm{FS3}} \wedge \mathrm{FID} \in \mathrm{FS3} \wedge \mathrm{FS2} \subseteq \mathrm{FS3}\ \}$
`v := c.value;`
`}`
$\{\ \exists \mathrm{FS3}.\ \mathtt{n} \to \mathrm{N} * \mathtt{v} \to \mathrm{VAL} * \alpha \mapsto \mathrm{NM}_{\mathrm{ID}}[\#\mathrm{text}_{\mathrm{TID}}[\mathrm{VAL}]_{\mathrm{FS1}}]_{\mathrm{FS3}} \wedge \mathrm{FS2} \subseteq \mathrm{FS3}\ \}$

A.1.5. Code for `fileCopy(source, target)`

The source code for, and associated proof of, the `fileCopy` command of section 6.3.4 is:

$$\left\{ \begin{array}{c} path(\texttt{path}, \mathrm{P}_1, \mathrm{A}) \wedge path(\texttt{target}, \mathrm{P}_2, \mathrm{C}, \mathrm{D}) \\ \wedge\ \alpha^{\mathrm{P}_1} \mapsto \mathrm{A} : \mathrm{I}_1 * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} \wedge can\_create(\mathrm{A})] * \mathrm{I}_1 \overset{\mathrm{F}}{\mapsto} \mathrm{SD} * E \end{array} \right\}$$

```
ret := fileCopy(source, target) ≜ local s, sourceH, targetH, buffer, c {
  buffer := malloc(4096)
  sourceH := open(source, 0);
  c := read(sourceH, buffer, 4096);
```

$$\left\{ \begin{array}{c} \exists\ \mathrm{SH}, \mathrm{TH}, \mathrm{I}_2, \mathrm{SD}_2, \mathrm{SB}, \mathrm{SD}_3. \\ \mathrm{SD} = \mathrm{SD}_2 \cdot \mathrm{SB} \cdot \mathrm{SD}_3 \wedge ((|\mathrm{SB}| < 4096 \wedge \mathrm{SD}_3 = \epsilon) \vee |\mathrm{SB}| = 4096) \\ \wedge\ \texttt{sourceH} \to \mathrm{SH} * \texttt{targetH} \to \mathrm{TH} * \texttt{buffer} \to \mathrm{BUF} * \texttt{c} \to |\mathrm{SB}| \\ * \ \mathrm{BUF} - 1 \mapsto 4096 * ncells(\mathrm{BUF}, |\mathrm{SB}|, \mathrm{SB}) * ncells(\mathrm{BUF} + |\mathrm{SB}|, 4096 - |\mathrm{SB}|, -) \\ * \ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}_1 * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{A} : \mathrm{I}_2] * \mathrm{I}_1 \overset{\mathrm{F}}{\mapsto} \mathrm{SD} \\ * \ E * \mathrm{I}_2 \overset{\mathrm{F}}{\mapsto} \mathrm{SD}_2 * \mathrm{SH} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}_1, |\mathrm{SD}_2 \cdot \mathrm{SB}|) * \mathrm{TH} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}_2, |\mathrm{SD}_2|) \end{array} \right\}$$

```
  while c = 4096
    write(targetH, buffer, 4096);
    c := read(sourceH, buffer, 4096);
  // We read less than 4096, so it was the last chunk
```

$$\left\{ \begin{array}{c} \exists\ \mathrm{SH}, \mathrm{TH}, \mathrm{I}_2, \mathrm{SD}_2, \mathrm{SB}, \mathrm{SD}_3. \\ \mathrm{SD} = \mathrm{SD}_2 \cdot \mathrm{SB} \cdot \mathrm{SD}_3 \wedge |\mathrm{SB}| < 4096 \wedge \mathrm{SD}_3 = \epsilon \\ \wedge\ \texttt{sourceH} \to \mathrm{SH} * \texttt{targetH} \to \mathrm{TH} * \texttt{buffer} \to \mathrm{BUF} * \texttt{c} \to |\mathrm{SB}| \\ * \ \mathrm{BUF} - 1 \mapsto 4096 * ncells(\mathrm{BUF}, |\mathrm{SB}|, \mathrm{SB}) * ncells(\mathrm{BUF} + |\mathrm{SB}|, 4096 - |\mathrm{SB}|, -) \\ * \ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}_1 * \beta^{\mathrm{P}_2} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{A} : \mathrm{I}_2] * \mathrm{I}_1 \overset{\mathrm{F}}{\mapsto} \mathrm{SD} \\ * \ E * \mathrm{I}_2 \overset{\mathrm{F}}{\mapsto} \mathrm{SD}_2 * \mathrm{SH} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}_1, |\mathrm{SD}_2 \cdot \mathrm{SB}|) * \mathrm{TH} \overset{\mathrm{PH}}{\mapsto} (\mathrm{I}_2, |\mathrm{SD}_2|) \end{array} \right\}$$

```
  s := write(targetH, buffer, c);
  close(sourceH); close(targetH); free(buffer)
} { ∃ I2.α^P1 ↦ A : I1 * β^P2 ↦ D[C + A : I2] * I1 ↦F SD * E * I2 ↦F SD }
```

A.2. Unique collapse to heap

Here, we provide the proof of theorem 3, showing that if an abstract heap collapses to a complete heap, the complete heap is unqiue. This result is *confluence*. It is well known, and typically proven via local confluence and Newman's lemma. Unfortunately, our system does not satisfy local confluence on all pre-abstract heaps. The pre-abstract heap $\mathbf{x} \mapsto \mathbf{y} * \mathbf{y} \mapsto \mathbf{x}$ has two possible collapses, $\mathbf{x} \mapsto \mathbf{x}$ and $\mathbf{y} \mapsto \mathbf{y}$. Neither can collapse further, but both are different. However, if $\mathbf{ph} \downarrow^* h$, then $\downarrow$ *is* locally confluent on $\mathbf{ph}$. We can thus

create a proof based upon *conditional* local confluence.

Lemma 26 (Conditional local confluence)**.** *Assume that, for all* $\mathbf{ph}, \mathbf{ph}_2 \in \textsc{PreAbsHeaps}$ *and* $h \in \textsc{StructHeaps}$*,* $\mathbf{ph} \downarrow^* h$ *and* $\mathbf{ph} \downarrow \mathbf{ph}_2$*. Then,* $\mathbf{ph}_2 \downarrow^* h$*.*

Proof. Since $\mathbf{ph} \downarrow \mathbf{ph}_2$, we know that $\#(\mathbf{ph}) > 0$. Moreover, since $\mathbf{ph} \downarrow^* h$, we know that $\exists \mathbf{ph}_1.\ \mathbf{ph} \downarrow \mathbf{ph}_1 \downarrow^* h$. Moreover, by the definition of single-step collapse, $\mathrm{dom}(\mathbf{ph}) \setminus \mathrm{dom}(\mathbf{ph}_1) = \{\mathbf{x}\}$ and $\mathrm{dom}(\mathbf{ph}) \setminus \mathrm{dom}(\mathbf{ph}_2) = \{\mathbf{y}\}$ (that is, the collapse to $\mathbf{ph}_1$ used abstract address $\mathbf{x}$, and the collapse to $\mathbf{ph}_2$ used abstract address $\mathbf{y}$).

Let $\#(\mathbf{ph}) = |\mathrm{dom}(\mathbf{ph}) \cap \textsc{StructAddrs}|$ (the function giving the number of structural addresses in the domain of some pre-abstract heap). We proceed by induction on $\#(\mathbf{ph})$.

1. $\#(\mathbf{ph}) = 0$: The result follows directly, as $\mathbf{ph} \not\downarrow$.
2. $\#(\mathbf{ph}) = n + 1$: We know that $\mathbf{ph} \downarrow \mathbf{ph}_1$ using hole $\mathbf{x}$ and $\mathbf{ph} \downarrow \mathbf{ph}_2$ using hole $\mathbf{y}$. If $\mathbf{x} = \mathbf{y}$, the result follows directly.

   Assume then, that $\mathbf{x} \neq \mathbf{y}$. We demonstrate that there exists some $\mathbf{ph}_3$ such that $\mathbf{ph}_1 \downarrow \mathbf{ph}_3$ and $\mathbf{ph}_2 \downarrow \mathbf{ph}_3$. With this result, we have $\mathbf{ph}_1 \downarrow^* h$, $\mathbf{ph}_1 \downarrow \mathbf{ph}_3$ and $\#(\mathbf{ph}_1) = n$. So, by the inductive hypothesis, we have $\mathbf{ph}_3 \downarrow^* h$. Hence, $\mathbf{ph}_2 \downarrow^* h$ as required.

   To prove this remaining result, examine the possible ways $\mathbf{ph}$ can be constructed such that $\mathbf{ph} \downarrow \mathbf{ph}_1$ and $\mathbf{ph} \downarrow \mathbf{ph}_2$. There must two abstract heap cells $\mathbf{x}, \mathbf{y} \in \mathrm{dom}(\mathbf{ph})$. There may exist at most two additional heap cells with addresses $\mathbf{a}_1, \mathbf{a}_2 \in \mathrm{dom}(\mathbf{ph})$ which contain $\mathbf{x}, \mathbf{y}$ as body addresses.

   In figure A.2, we consider all the permutations for the location of the body addresses $\mathbf{x}$ and $\mathbf{y}$. The rows indicate which heap addresses contains $\mathbf{x}$ and $\mathbf{y}$ as body addresses, and is labelled by a justification. In each case, this justification either demonstrates the existence of $\mathbf{ph}_3$ or shows that the case is impossible. The justifications are:

   Quazi-commutivity: In these cases, construct $\mathbf{ph}_3$ by collapsing using $\mathbf{y}$ on $\mathbf{ph}_1$ and using $\mathbf{x}$ on $\mathbf{ph}_2$. The result is such that both collapses are into exactly one of the cells $\mathbf{a}_1$ or $\mathbf{a}_2$, and the equality of the results follows by quazi-commutativity of structurally addressed data.

   Quazi-associativity: We construct $\mathbf{ph}_3$ as in the quazi-commutitive case. However, here, the equality is given by quazi-associativity of structurally addressed data.

Heap cells that may be $\mathbf{x}$ *and* $\mathbf{y}$

Position of $\mathbf{x}$ *and* $\mathbf{y}$ *addresses*

	$\mathbf{ph}(\mathbf{a}_1)$	$\mathbf{ph}(\mathbf{a}_2)$	$\mathbf{ph}(\mathbf{x})$	$\mathbf{ph}(\mathbf{y})$	Justification
1	$\mathbf{x}, \mathbf{y}$				✓, Quazi-commutativity
2	$\mathbf{x}$	$\mathbf{y}$			✓, Pick alternate
3	$\mathbf{x}$		$\mathbf{y}$		✓, Quazi-associativity
4	$\mathbf{x}$			$\mathbf{y}$	×, $\mathbf{ph} \not\downarrow \mathbf{ph}_2$
5	$\mathbf{y}$	$\mathbf{x}$			✓, Pick alternate
6		$\mathbf{x}, \mathbf{y}$			✓, Quazi-commutativity
7		$\mathbf{x}$	$\mathbf{y}$		✓, Quazi-associativity
8		$\mathbf{x}$		$\mathbf{y}$	×, $\mathbf{ph} \not\downarrow \mathbf{ph}_2$
9	$\mathbf{y}$		$\mathbf{x}$		×, $\mathbf{ph} \not\downarrow \mathbf{ph}_1$
10		$\mathbf{y}$	$\mathbf{x}$		×, $\mathbf{ph} \not\downarrow \mathbf{ph}_1$
11			$\mathbf{x}, \mathbf{y}$		×, $\mathbf{ph} \not\downarrow \mathbf{ph}_1$
12			$\mathbf{x}$	$\mathbf{y}$	×, $\mathbf{ph} \not\downarrow \mathbf{ph}_1$
13	$\mathbf{y}$			$\mathbf{x}$	✓, Quazi-associativity
14		$\mathbf{y}$		$\mathbf{x}$	✓, Quazi-associativity
15			$\mathbf{y}$	$\mathbf{x}$	×, No final heap
16				$\mathbf{x}, \mathbf{y}$	×, $\mathbf{ph} \not\downarrow \mathbf{ph}_2$

Figure A.2.: Case analysis for proof of lemma 26. The four central columns indicate the heap addresses in which the given body addresses are found.

Pick alternate: In these cases, construct $\mathbf{ph}_3$ by collapsing using $\mathbf{y}$ on $\mathbf{ph}_1$ and using $\mathbf{x}$ on $\mathbf{ph}_2$. The result is such that $\mathbf{x}$ collapsed into $\mathbf{a}_1$ and $\mathbf{y}$ collapsed into $\mathbf{a}_2$ (or vice versa), and the result follows directly.

$\mathbf{ph} \not\downarrow \mathbf{ph}_2$: This permutation cannot occur, as $\mathbf{ph}$ could never use $\mathbf{y}$ to collapse to $\mathbf{ph}_2$.

$\mathbf{ph} \not\downarrow \mathbf{ph}_1$: This permutation cannot occur, as $\mathbf{ph}$ could never use $\mathbf{x}$ to collapse to $\mathbf{ph}_1$.

No final heap: This permutation cannot occur, as $\mathbf{ph}_1$ must satisfy $\mathbf{y} \in \mathbf{ph}_1(\mathbf{y})$. Evidently, there is exactly one $\mathbf{y} \in \mathrm{dom}(\mathbf{ph}_1)$. The only way to remove $\mathbf{y}$ from the body of $\mathbf{ph}_1(\mathbf{y})$ is to collapse a $\mathbf{y}$-addressed cell into it, yet collapse will never perform this here by definition. Ergo, there is no $\mathbf{ph}_n$ such that $\mathbf{ph}_1 \downarrow^* \mathbf{ph}_n$ and $\nexists \mathbf{a}.\ \mathbf{y} \in \mathbf{ph}_n(\mathbf{a})$. Yet, h satisfies this second property, and $\mathbf{ph}_1 \downarrow^* h$. Contradiction.

□

Given this local confluence, we can prove a many-step version.

Lemma 27 (Conditional confluence)**.** *Assume that, for all* $\mathbf{ph}, \mathbf{ph}_1 \in \textsc{PreAbsHeaps}$ *and* $h \in \textsc{StructHeaps}$*,* $\mathbf{ph} \downarrow^* h$ *and* $\mathbf{ph} \downarrow^* \mathbf{ph}_1$*. Then,* $\mathbf{ph}_1 \downarrow^* h$*.*

Proof. Again, we proceed by induction on the number of abstract addresses in $\mathbf{ph}$, $\#(\mathbf{ph})$.

- $\#(\mathbf{ph}) = 0$: By definition, $\mathbf{ph}_1 = h$, and the result follows.
- $\#(\mathbf{ph}) = n + 1$: Then, there exists $\mathbf{ph}_1'$ such that $\mathbf{ph} \downarrow \mathbf{ph}_1' \downarrow^* \mathbf{ph}_1$. Conditional confluence (lemma 26) ensures that $\mathbf{ph}_1' \downarrow^* h$. Now, $\#(\mathbf{ph}_1') = n$, $\mathbf{ph}_1' \downarrow^* h$ and $\mathbf{ph}_1' \downarrow^* \mathbf{ph}_2$. The inductive hypothesis provides the result.

□

Theorem 5 (Unique collapse)**.** *For all* $\mathbf{ph}_1 \in \textsc{PreAbsHeaps}$*, if there exists* $h \in \textsc{StructHeaps}$ *such that* $\mathbf{ph}_1 \downarrow^* h$*, then for all* $\mathbf{ph}_2 \in \textsc{PreAbsHeaps}$*,* $\mathbf{ph}_1 \downarrow^* \mathbf{ph}_2$ *and* $\mathbf{ph}_2 \not\downarrow$ *implies* $\mathbf{ph}_2 = h$*.*

Proof. By conditional confluence (lemma 27), if $\mathbf{ph}_1 \downarrow^* h$ and $\mathbf{ph}_1 \downarrow^* \mathbf{ph}_2$, then $\mathbf{ph}_2 \downarrow^* h$. We also have that $\mathbf{ph}_2 \not\downarrow$. The only way this is possible is if $\mathbf{ph}_2 = h$. □

This result allows us to show that abstract heaps are unambiguous and acyclic.

Lemma 28 (Abstract heaps are unambiguous)**.** *All* $\mathbf{h} \in \textsc{AbsHeaps}$ *are unambiguous, in that there are no* $\mathbf{x} \in \textsc{StructAddrs}$ *such that there exists* $\mathbf{a}_1, \mathbf{a}_2 \in \textsc{Addrs}$ *with* $\mathbf{x} \in \mathbf{h}(\mathbf{a}_1)$ *and* $\mathbf{x} \in \mathbf{h}(\mathbf{a}_2)$*.*

Proof. We proceed by contradiction. Assume $\mathbf{h}$ is ambiguous. Therefore, there exists $\mathbf{a}_1$ and $\mathbf{a}_2$ with $\mathbf{x} \in \text{addrs}(\mathbf{h}(\mathbf{a}_1))$ and $\mathbf{x} \in \text{addrs}(\mathbf{h}(\mathbf{a}_2))$. By the definition of $\mathbf{h}$, there exists an extension $\mathbf{ph} \in \textsc{PreAbsHeaps}$ and machine heap h such that $\mathbf{h} \sqcup \mathbf{ph} \downarrow^* h$.

By the definition of heaps, there can be no $\mathbf{a} \in \text{dom}(h)$ such that $\mathbf{x} \in \text{addrs}(h(\mathbf{a}))$ (as heaps contain no structural addresses). Therefore, there must be $\mathbf{x} \in \text{dom}(\mathbf{h} \sqcup \mathbf{ph})$, and some step in the chain of collapses that collapsed the $\mathbf{x}$ addressed heap cell into a value. However, by the definition of collapse, this is not possible, as there is never a *unique* $\mathbf{a}$ containing $\mathbf{x}$. Ergo, no matter what chain of collapses is used, it must eventually reach a pre-abstract heap $\mathbf{ph}'$ such that $\mathbf{h} \sqcup \mathbf{ph} \downarrow^* \mathbf{ph}', \mathbf{ph}' \not\downarrow$ and $\mathbf{x} \in \text{dom}(\mathbf{ph}')$. But, by theorem 3, as $\mathbf{h} \sqcup \mathbf{ph} \downarrow^* h$ so $\mathbf{ph}' \downarrow^* h$, which is impossible. Ergo, $\mathbf{h}$ was not ambiguous. □

Lemma 29 (Abstract heaps are acyclic)**.** *All* $\mathbf{h} \in \textsc{AbsHeaps}$ *are acyclic, in that there are no structural addresses* $\mathbf{x} \in \textsc{StructAddrs}$ *and (possibly empty) chain of single-step collapse steps such that* $\mathbf{h} \downarrow^* \mathbf{h}_n$ *with* $\mathbf{x} \in \mathit{addrs}(\mathbf{h}_n(\mathbf{x}))$*.*

Proof. We proceed by contradiction. Assume $\mathbf{h}$ is cyclic. By definition, there is some concrete structured heap h and pre-abstract heap $\mathbf{ph}$ such that $\mathbf{h} \sqcup \mathbf{ph} \downarrow^* h$, where h contains no structural addresses. By the definition of cyclicity, there is some $\mathbf{h}_n$ such that $\mathbf{x} \in \text{addrs}(\mathbf{h}_n(\mathbf{x}))$. There must exist $\mathbf{h}_m$ such that $\mathbf{h}_n \downarrow^* \mathbf{h}_m$, $\mathbf{h}_m \not\downarrow$, with $\mathbf{x} \in \text{addrs}(\mathbf{h}_m(\mathbf{x}))$ as, by definition, collapse will never compress an abstract heap cell into itself and, by construction, the domain can never contain an addition $\mathbf{x}$ by which collapse can remove the body address $\mathbf{x}$. By theorem 3, it must be the $\mathbf{h}_m = h$, which cannot be, as $\mathbf{h}_m$ contains structural addresses. Ergo, $\mathbf{h}$ was not cyclic. □

A.3. Promise- and obligation-carrying data as structural separation algebras

This section provides the proof that promise- and obligation-carrying data is a structural addressing algebra. The proofs proceed as in the promises case given in section 5.2.1.

Definition 144 (Promise- and obligation-carrying data algebra)**.** Let $(\textsc{StructAddrs}, \textsc{Data}, \text{addrs}, \text{comp})$ be some structural addressing algebra (definition 42). Let $(\textsc{StructAddrs}_{\Subset}, \textsc{Data}_{\Subset}, \text{addrs}_{\Subset}, \text{comp}_{\Subset})$ be the associated underlying data algebra (definition 90). Let $\textsc{PromObsData}$ be the set of promise- and obligation-carrying data defined using these previous two algebras. Then, the associated *promise- an obligation-carrying data algebra* is defined as:

$$(\textsc{StructAddrs}_O, \textsc{PromObsData}, \text{addrs}_O, \text{comp}_O)$$

where

1. The structural addresses are those of the underlying data: $\textsc{StructAddrs}_P = \textsc{StructAddrs}_{\Subset}$.
2. The addresses function is the addresses function of the underlying data, applied to the underlying data component of a promise-carrying data $\text{addrs}(\mathbf{pod}) = \text{addrs}_{\Subset}(\mathbf{pod}\downarrow_1)$.

3. The compression function comp_P is that defined on PROMOBSDATA, given in definition 139.

When unambiguous, $\text{comp}_O(\mathbf{x}, \mathbf{pod}_1, \mathbf{pod}_2)$ is written $\mathbf{pod}_1 \circledast \mathbf{pod}_2$, and addrs_O is written addrs. The data of the underlying algebra are equivalent to data with no promises and obligations $d \in \text{DATA}_{\Cap} = (d, \emptyset, \emptyset)$.

This algebra inherits many useful properties from the underlying structural addressing algebra. However, we must first ensure that the compression function preserves the well-formedness of the richer data we are using. For the lemmas, assume a structural addressing algebra (STRUCTADDRS, DATA, addrs, comp) for the promises and obligations, and the associated underlying data algebra ((STRUCTADDRS$_O$, PROMOBSDATA, addrs$_O$, comp$_O$).

Lemma 30 (Compression is closed on PROMOBSDATA)**.** *Let* $\mathbf{pod}_1, \mathbf{pod}_2 \in$ PROMOBSDATA*,* $\mathbf{x} \in$ STRUCTADDRS *and* $\mathbf{pod}_1 \circledast \mathbf{pod}_2$ *be defined. Then,* $\mathbf{pod}_1 \circledast \mathbf{pod}_2 \in$ PROMOBSDATA*.*

Proof. Assume $\mathbf{pod}_1 = (d_1, \Pi_1, \Omega_1)$ and $\mathbf{pod}_2 = (d_2, \Pi_2, \Omega_2)$. The result of $\mathbf{pod}_1 \circledast \mathbf{pod}_2$ is:

$$(d_1 \circledast d_2, (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap), (\Omega_1 - \mathbf{x}) \cup (\Omega_2 - \Cap)))$$

As per the definitions of promises, obligations and promise- and obligation-carrying data (definitions 94, 137 and 138 respectively), this will be valid promise- and obligation-carrying data if: 1) there are no duplicate promise or obligation addresses; 2) every promise or obligation is addressed to either $\Cap$ or a body address within the data; and 3) every obligation is fulfilled.

Note first that, by the addressing properties of the underlying structural addressing algebras, the set $\text{addrs}(d_1 \circledast d_2)$ contains the addresses $\text{addrs}((d_1 \setminus \{\mathbf{x}\})) \cup \text{addrs}(d_2)$. Call this the *address containment property.*

1. Proceed by contradiction. Assume an address a that identifies two promises: $\{(a, \overline{d}_1), (a, \overline{d}_2)\} \subseteq (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap)$.

   Notice that $a \neq \Cap$, as by construction $\Cap$ is removed from Π_2 in the compression, and Π_1 is well-formed. Ergo $a \in$ STRUCTADDRS $\setminus \{\Cap\}$. Therefore, by well-formedness of $\mathbf{pod}_1$ and $\mathbf{pod}_2$, $a \in \text{addrs}_O(\mathbf{pod}_1)$ or $a \in \text{addrs}_O(\mathbf{pod}_2)$. Assume $a = \mathbf{x}$. Then it cannot be duplicated, as $\mathbf{x}$ is removed from Π_1 in the union and Π_2 is well-formed. However, it is also the case that $a \neq \mathbf{x}$. This follows from the address containment properties, which show that $\text{addrs}(d_1) \cup \text{addrs}(d_2) \setminus \{\mathbf{x}\} = \emptyset$.

The case for obligations is similar.

2. Proceed by contradiction. Assume there is a promise in $(a, \overline{d}) \in (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Rightarrow)$ such that $a \neq \Rightarrow$ and $a \notin \text{addrs}(d_1 \circledast d_2)$. By definition, either $a \in \Pi_1$ or $a \in \Pi_2$. By the well-formedness of $\mathbf{pod}_1$ and $\mathbf{pod}_2$, either $a \in \text{addrs}_O(\mathbf{pod}_1)$, or $a \in \text{addrs}_O(\mathbf{pod}_2)$. If $a \neq \mathbf{x}$, then by the address containment property, it must be that $a \in \text{addrs}(d_1 \circledast d_2)$, which contradicts its existence.

   If $a = \mathbf{x}$, then by the construction of $(\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Rightarrow)$, a addresses a promise in Π_2. Ergo, $a \in \text{addrs}_O(\mathbf{pod}_2)$ by well-formedness. By the address preservation property of structural addressing algebras, $a \in \text{addrs}(d_1 \circledast d_2)$, which again contradicts its existence.

   The case for obligations is similar.

3. Proceed by contradiction. Assume there is an unfulfilled obligation $(a, \overline{d}) \in (\Omega_1 - \mathbf{x}) \cup (\Omega_2 - \Rightarrow)$. Assume first that $a = \Rightarrow$. Then, by construction, the obligation must be $(\Rightarrow, \underline{d}) \in \Omega_1$. As the obligation is not fulfilled, it must be that $\downarrow (d_1 \circledast d_2, (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Rightarrow)) \not\subseteq \underline{d}$. However, we have $\downarrow (d_1 \circledast d_2, (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Rightarrow)) \subseteq \downarrow (d_1, \Pi_1)$. This must be true, as either $\mathbf{pod}_1$ had no promise from $\mathbf{x}$ for d_2, and so the body closure considered *all* possibilities, or $\mathbf{pod}_1$ did have a promise, which was fulfilled by the obligations of $\mathbf{pod}_2$. Moreover, $\downarrow (d_1, \Pi_1) \subseteq \underline{d}$, as $\mathbf{pod}_1$ is well-formed. Ergo the obligation must be met, as $\subseteq$ is transitive.

   Assume now that $a \neq \Rightarrow$. Then, via the proof of case 2) above, it must be that the obligation is $(\mathbf{x}, \overline{d})$ for some $\mathbf{y} \in \text{addrs}(d_1 \circledast d_2)$. As the obligation is not fulfilled, is must be that $\circlearrowleft_{\mathbf{y}} ((d_1 \circledast d_2, (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Rightarrow))) \not\subseteq \overline{d}$. However, by reasoning similar to the above, this cannot be: $\circlearrowleft_{\mathbf{y}} ((d_1 \circledast d_2, (\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Rightarrow))) \subseteq \circlearrowleft_{\mathbf{y}} ((d_1, \Pi_1))$, as the body closure around $\mathbf{x}$ (even if vacuous) must contain at least the actual data d_2.

□

Of the underlying data algebra properties, the following are maintained.

Lemma 31 (Value containment)**.** *Given* Data *that is the underlying structural addressing algebra for the promise- and obligation-carrying data algebra, and* Values *that is the set of values underlying* Data*,* Values $\subseteq$ Data*.*

Proof. Follows from the lift of data without promises or obligations to be equal to promise-carrying data, and the value containment property of the underlying data. □

Lemma 32 (Address properties)**.** *For all* $\mathbf{pod}_1, \mathbf{pod}_2 \in \textsc{PromObsData}$ *and* $\mathbf{x} \in \textsc{StructAddrs}_O$*, if* $\mathbf{pod}_1 \circledast{\mathbf{x}} \mathbf{pod}_2$ *is defined then:*

1. Containment*:* $\mathbf{x} \in \mathit{addrs}_O(\mathbf{pod}_1)$.
2. Non-overlap*:* $\mathit{addrs}_O(\mathbf{pod}_1) \cap \mathit{addrs}_O(\mathbf{pod}_2) \subseteq \{\mathbf{x}\}$
3. Preservation*:* $(\mathit{addrs}_O(\mathbf{pod}_1) \setminus \{\mathbf{x}\}) \cup \mathit{addrs}_O(\mathbf{pod}_2) = \mathit{addrs}_O(\mathbf{pod}_1 \circledast{\mathbf{x}} \mathbf{pod}_2)$

Proof. These follow directly from the addresses function being defined in terms of the underlying addresses function, which is applied to the underlying data. □

Lemma 33 (Compression quasi-associates)**.** *Let* $\mathbf{pod}_1, \mathbf{pod}_2, \mathbf{pod}_3 \in \textsc{PromObsData}$ *and* $\mathbf{x}, \mathbf{y} \in \textsc{StructAddrs}$*, with* $\mathbf{pod}_1 = (d_1, \Pi_1, \Omega_1), \mathbf{pod}_2 = (d_2, \Pi_2, \Omega_2), \mathbf{pod}_3 = (d_3, \Pi_3, \Omega_3)$*. If* $\mathbf{y} \in \mathit{addrs}_O(\mathbf{pod}_2)$ *and either* $\mathbf{y} \notin \mathit{addrs}_O(\mathbf{pod}_1)$ *or* $\mathbf{y} = \mathbf{x}$*, then* $\mathbf{pod}_1 \circledast{\mathbf{x}}(\mathbf{pod}_2 \circledast{\mathbf{y}} \mathbf{pod}_3) = (\mathbf{pod}_1 \circledast{\mathbf{x}} \mathbf{pod}_2) \circledast{\mathbf{y}} \mathbf{pod}_3$.

Proof. Assume that $\mathbf{x} \neq \mathbf{y}$ (the case where $\mathbf{x} = \mathbf{y}$ is similar). Assume further that the left-hand side, $\mathbf{pod}_1 \circledast{\mathbf{x}}(\mathbf{pod}_2 \circledast{\mathbf{y}} \mathbf{pod}_3)$ is defined. Then, it must equal:

$$\left(\begin{array}{l} d_1 \circledast{\mathbf{x}}(d_2 \circledast{\mathbf{y}} d_3), \\ \end{array} \begin{array}{l} (\Pi_1 - \mathbf{x}) \cup ((\Pi_2 - \mathbf{y}) \cup (\Pi_3 - \Cap)) - \Cap, \\ (\Omega_1 - \mathbf{x}) \cup ((\Omega_2 - \mathbf{y}) \cup (\Pi_3 - \Cap)) - \Cap \end{array} \right)$$

The removal of a promise or obligation from a set is idempotent. Therefore, we have:

$$\left(\begin{array}{l} d_1 \circledast{\mathbf{x}}(d_2 \circledast{\mathbf{y}} d_3), \\ \end{array} \begin{array}{l} (\Pi_1 - \mathbf{x}) \cup ((\Pi_2 - \mathbf{y} - \Cap) \cup (\Pi_3 - \Cap)), \\ (\Omega_1 - \mathbf{x}) \cup ((\Omega_2 - \mathbf{y} - \Cap) \cup (\Pi_3 - \Cap)) \end{array} \right)$$

As $\mathbf{x} \neq \mathbf{y}$, there are no promises named $\mathbf{y}$ in Π_1 (similarly for the obligations). Moreover, as standard set union associates, we have:

$$\left(\begin{array}{l} (d_1 \circledast{\mathbf{x}} d_2) \circledast{\mathbf{y}} d_3, \\ \end{array} \begin{array}{l} ((\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap)) - \mathbf{y} \cup (\Pi_3 - \Cap), \\ ((\Omega_1 - \mathbf{x}) \cup (\Omega_2 - \Cap)) - \mathbf{y} \cup (\Omega_3 - \Cap), \end{array} \right)$$

which is equal to $(\mathbf{pod}_1 \circledast{\mathbf{x}} \mathbf{pod}_2) \circledast{\mathbf{y}} \mathbf{pod}_3$ (data quasi-associativity holds by the definition of structural addressing algebras). Ergo, if the left is defined, the right is defined and equal to it. □

Lemma 34 (Compression quasi-commutes)**.** *Let* $\mathbf{pod}_1, \mathbf{pod}_2, \mathbf{pod}_3 \in \textsc{PromData}$ *and* $\mathbf{x}, \mathbf{y} \in \textsc{StructAddrs}$*, with* $\mathbf{pod}_1 = (d_1, \Pi_1, \Omega_1), \mathbf{pod}_2 = (d_2, \Pi_2, \Omega_2), \mathbf{pod}_1 = (d_3, \Pi_3, \Omega_3)$*. If* $\mathbf{x} \notin \mathit{addrs}_O(\mathbf{pod}_3)$ *and* $\mathbf{y} \notin \mathit{addrs}_O(\mathbf{pod}_2)$*, then* $(\mathbf{pod}_1 \circledast{\mathbf{x}} \mathbf{pod}_2) \circledast{\mathbf{y}} \mathbf{pod}_3 = (\mathbf{pod}_1 \circledast{\mathbf{y}} \mathbf{pod}_3) \circledast{\mathbf{x}} \mathbf{pod}_2$.

Proof. Assume that the left hand side, $\mathbf{pod}_1 \circledast{\mathbf{x}} \mathbf{pod}_2 \circledcirc{\mathbf{y}} \mathbf{pod}_3$ is defined. Then, it must equal:

$$\left(\begin{array}{c} (d_1 \circledast{\mathbf{x}} d_2) \circledcirc{\mathbf{y}} d_3, \\ \end{array} \begin{array}{c} ((\Pi_1 - \mathbf{x}) \cup (\Pi_2 - \Cap)) - \mathbf{y} \cup (\Pi_3 - \Cap), \\ ((\Omega_1 - \mathbf{x}) \cup (\Omega_2 - \Cap)) - \mathbf{y} \cup (\Omega_3 - \Cap) \end{array} \right)$$

As Π_2 has no promise named $\mathbf{y}$ (and similarly for the obligations), we have:

$$\left(\begin{array}{c} (d_1 \circledast{\mathbf{x}} d_2) \circledcirc{\mathbf{y}} d_3, \\ \end{array} \begin{array}{c} ((\Pi_1 - \mathbf{x} - \mathbf{y}) \cup (\Pi_2 - \Cap)) \cup (\Pi_3 - \Cap), \\ ((\Omega_1 - \mathbf{x} - \mathbf{y}) \cup (\Omega_2 - \Cap)) \cup (\Omega_3 - \Cap) \end{array} \right)$$

As Π_3 has no promise named $\mathbf{x}$, we then have

$$\left(\begin{array}{c} (d_1 \circledast{\mathbf{x}} d_2) \circledcirc{\mathbf{y}} d_3, \\ \end{array} \begin{array}{c} ((\Pi_1 - \mathbf{y}) \cup (\Pi_3 - \Cap)) - \mathbf{x} \cup (\Pi_2 - \Cap), \\ ((\Omega_1 - \mathbf{y}) \cup (\Omega_3 - \Cap)) - \mathbf{x} \cup (\Omega_2 - \Cap) \end{array} \right)$$

which is $\mathbf{pod}_1 \circledcirc{\mathbf{y}} \mathbf{pod}_3 \circledast{\mathbf{x}} \mathbf{pod}_2$. □

The underlying data has a simple left identity property, as $\mathbf{x} \circledast{\mathbf{x}} d = d$ always. The addition of promises and obligations breaks this simple identity, simply by picking any d with a head promise. However, there are still identities, which can be seen via construction based upon the choice of d. We show the existence of left identities below; the right identities are demonstrated in a similar manner.

Lemma 35 (Existence of left identities)**.** *For all* $\mathbf{pod} \in$ PromObsData *and* $\mathbf{x} \in$ StructAddrs*, there exists an* $\mathbf{pod}_{\mathbf{x}} \in$ PromObsData *such that* $\mathbf{pod}_{\mathbf{x}} \circledast{\mathbf{x}} \mathbf{pod} = \mathbf{pod}$.

Proof. Assume $\mathbf{pod} = (d, \Pi, \Omega)$. We will construct of $\mathbf{pod}_{\mathbf{x}} = (\mathbf{x}, \Pi_{\mathbf{x}}, \Omega_{\mathbf{x}})$ by picking $\Pi_{\mathbf{x}}$ and $\Omega_{\mathbf{x}}$ based on the contents of Π and Ω. Without futher information, the compression is:

$$(\mathbf{x} \circledast{\mathbf{x}} d, (\Pi_{\mathbf{x}} - \mathbf{x}) \cup (\Pi - \Cap), (\Omega_{\mathbf{x}} - \mathbf{x}) \cup (\Omega - \Cap))$$

Assume that $(\Cap, \overline{d}) \in \Pi$ and $(\Cap, \underline{d}) \in \Omega$. In this case, we pick $\Pi_{\mathbf{x}} = \{(\mathbf{x}, \underline{d}), (\Cap, \overline{d})\}$ and $\Omega = \{(\mathbf{x}, \overline{d}), (\Cap, \underline{d})\}$. This is well-formed by construction. By calculation, using the identity property of data:

$$\begin{array}{rl} & (\mathbf{x} \circledast{\mathbf{x}} d, (\Pi_{\mathbf{x}} - \mathbf{x}) \cup (\Pi - \Cap), (\Omega_{\mathbf{x}} - \mathbf{x}) \cup (\Omega - \Cap)) \\ = & (d, (\{(\mathbf{x}, \underline{d}), (\Cap, \overline{d})\} - \mathbf{x}) \cup (\Pi - \Cap), (\{(\mathbf{x}, \overline{d}), (\Cap, \underline{d})\} - \mathbf{x}) \cup (\Omega - \Cap)) \\ = & (d, \{(\Cap, \overline{d})\} \cup (\Pi - \Cap), \{(\Cap, \underline{d})\} \cup (\Omega - \Cap)) \\ = & (d, \Pi, \Omega) \end{array}$$

as required. The cases where **pod** has either no head promise or head obligations are similar. □

The identity properties were used by structural addressing algebras to show that the choice of hole is irrelevant. One could always compress another structural address in, so that $d \circledast \mathbf{y}$ was defined if $\mathbf{y} \notin \text{addrs}(d)$. Without the simple identity properties, we cannot prove this directly. However, we can still show that the choice of address does not matter.

Lemma 36 (Address irrelevance)**.** *For all* $\mathbf{pod}_1, \mathbf{pod}_2 \in$ PromObsData *and* $\mathbf{x} \in \mathit{addrs}_O(\mathbf{pod}_1)$, $\mathbf{y} \notin \mathit{addrs}_O(\mathbf{pod}_2)$*, if* $\mathbf{pod}_1 \circledast \mathbf{pod}_2$ *is defined, there exists* $\mathbf{pod}_\mathbf{y}$ *such that* $\mathbf{pod}_1 \circledast \mathbf{pod}_\mathbf{y} \textcircled{y} \mathbf{pod}_2 = \mathbf{pod}_1 \circledast \mathbf{pod}_2$.

Proof. Knowing that $\mathbf{pod}_1 \circledast \mathbf{pod}_2$, assume that $\mathbf{pod}_1 = (d_1, \Pi_1, \Omega_1)$ and $\mathbf{pod}_2 = (d_2, \Pi_2, \Omega_2)$. We will consider the case where $(\mathbf{x}, \underline{d}_1) \in \Pi_1$, $(\Cap, \underline{d}_2) \in \Omega_2$, $(\Cap, \overline{d}_2) \in \Pi_2$ and $(\mathbf{x}, \overline{d}_1) \in \Omega_1$, (ergo, $\underline{d}_2 \subseteq \underline{d}_1$, $\overline{d}_1 \subseteq \overline{d}_2$). The cases without head or body obligations to one or the other are similar.

We construct $\mathbf{pod}_\mathbf{x}$ in a similar fashion to building the identities for data, by giving it the promises and obligations that match the expectations of $\mathbf{pod}_1$ and $\mathbf{pod}_2$. Pick $\mathbf{pod}_\mathbf{y} = (\mathbf{y}, \Pi_\mathbf{y}, \Omega_\mathbf{y})$ where $\Pi_\mathbf{y} = \{(\Cap, \overline{d}_1), (\mathbf{y}, \underline{d}_1)\}$ and $\Omega_\mathbf{y} = \{(\Cap, \underline{d}_1), (\mathbf{y}, \overline{d}_1)\}$. The result then follows by calculation, as in lemma 35. □

If our promise- and obligation-carrying data is a structural addressing algebra, we know it is safe to use as the data for structural separation logic. We can satisfy the requirements definition 42 as:

1. *Value containment*: True by lemma 31.
2. *Unaddressed values*: Satisfied by the lift of the addresses function in definition 144 and the underlying data.
3. *Address properties*: Satisfied by lift of the addresses function in definition 144, and the underlying data properties.
4. *Identity*: True by lemma 35.
5. *Arbitrary addresses*: True via lemma 36.
6. *Compression left-cancellativity*: True by definition of compression and underlying data.
7. *Compression quasi-associativity*: True by lemma 33.
8. *Compression quasi-commutativity*: True by lemma 34.

Therefore, given any abstract addressing algebra, we can construct a promise- and obligation-carrying structure, which by the lemmas above will be a weak structural addressing algebra. This is then a foundation on which we can build abstract heaps.

Corollary 2 (Abstract heaps with promises and obligations are sound)**.** *The construction of abstract heaps using promises and obligations on data is sound.*

Proof. As results in chapter 3. □

Bibliography

[1] POSIX.1-2008, IEEE Std 1003.1-2008, The Open Group Base Specifications Issue 7.

[2] Webkit bug 76591 (https://bugs.webkit.org/show_bug.cgi?id=76591), January 2012.

[3] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a File System Implementation. In *LNCS: Formal Methods and Software Engineering*. 2004.

[4] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[5] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In *Computer Aided Verification*, pages 119–122. Springer, 2010.

[6] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods*, pages 1–20. Springer, 2004.

[7] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, pages 115–137. Springer, 2006.

[8] Berners-Lee. World Wide Web: Proposal for a HyperText Project. 1990.

[9] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering, IEEE Transactions on*, 21(10):785–798, 1995.

[10] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM, 2005.

[11] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[12] S. Brookes. A semantics for concurrent separation logic. *CONCUR 2004-Concurrency Theory*, pages 16–34, 2004.

[13] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7(23-50):3, 1972.

[14] C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjunct elimination in context logic for trees. *Programming Languages and Systems*, pages 255–270, 2007.

[15] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *ACM SIGPLAN Notices*, volume 40, pages 271–282. ACM, 2005.

[16] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In *ACM SIGPLAN Notices*, volume 42, pages 123–134. ACM, 2007.

[17] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 366–378. IEEE, 2007.

[18] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, pages 140–155. Springer, 1998.

[19] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. A simple abstraction for complex concurrent indexes. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 845–864. ACM, 2011.

[20] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[21] T. Dinsdale-Young. *Abstract data and local reasoning.* PhD thesis, Imperial College London (University of London), 2011.

[22] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. Technical report, URL https://sites.google.com/site/viewsmodel, 2012.

[23] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. *ECOOP 2010–Object-Oriented Programming*, pages 504–528, 2010.

[24] D. Distefano. Attacking large industrial code with bi-abductive inference. *Formal Methods for Industrial Critical Systems*, pages 1–8, 2009.

[25] D. Distefano and M. J. Parkinson J. jStar: Towards practical verification for Java. *ACM Sigplan Notices*, 43(10):213–226, 2008.

[26] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. *Programming Languages and Systems*, pages 363–377, 2009.

[27] ECMAScript, ECMA and European Computer Manufacturers Association and others. ECMAScript language specification.

[28] K. Fisher, N. Foster, D. Walker, and K. Q. Zhu. Forest: A language and toolkit for programming with filestores. ICFP '11, 2011.

[29] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[30] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. *Engineering of Complex Computer Systems, IEEE International Conference*, 2007.

[31] L. Freitas, K. Mokos, and J. Woodcock. Verifying the CICS File Control API with Z/Eves: An Experiment in the Verified Software Repository. In *ICECCS*, 2007.

[32] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the verification grand challenge: A roadmap. In *ICECCS*, 2008.

[33] A. Galloway, G. Lüttgen, J. Mühlberg, and R. Siminiceanu. Model-Checking the Linux Virtual File System. In *LNCS: Verification, Model Checking, and Abstract Interpretation*. 2009.

[34] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–44. ACM, 2012.

[35] P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. DOM: Towards a formal specification. In *Proceedings of the ACM SIGPLAN workshop on Programming Language Technologies for XML (PLAN-X), California, USA*, 2008.

[36] P. Gardner, G. Smith, M. Wheelhouse, and U. D. Zarfaty. Local Hoare reasoning about DOM. In *Proceedings of the twenty-seventh ACM*

SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 261–270. ACM, 2008.

[37] P. Gardner, G. Smith, and A. Wright. Local reasoning about mashups. In *Theory workshop at the 3rd International Conference on Verified Software: Theories, Tools and Experiments*, volume 1, page 9, 2010.

[38] P. Gardner and M. Wheelhouse. Small specifications for tree update. *Web Services and Formal Methods*, pages 178–195, 2010.

[39] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[40] X. S. Group. XML schema, W3C recommendation, 2001.

[41] W. H. Hesselink and M. Lali. Formalizing a hierarchical file system. *REFINE 2009*, 2009.

[42] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[43] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *Proc. 3rd International Workshop on the World Wide Web and Databases, WebDB00*, volume 1997, 2000.

[44] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM SIGPLAN Notices*, volume 36, pages 14–26. ACM, 2001.

[45] B. Jacobs and F. Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[46] J. Jensen and L. Birkedal. Fictional separation logic. *Programming Languages and Systems*, pages 377–396, 2012.

[47] C. B. Jones. Specification and design of (parallel) programs. *Information processing*, 83(9):321, 1983.

[48] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. *Form. Asp. Comput.*, 2007.

[49] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[50] C. Morgan and B. Sufrin. Specification of the UNIX Filing System. *Software Engineering, IEEE Transactions on*, 1984.

[51] G. Ntzik. *Reasoning about POSIX File Systems (unconfirmed)*. PhD thesis, Imperial College London, To appear.

[52] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, pages 1–19. Springer, 2001.

[53] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, pages 215–244, 1999.

[54] O. OMGIDL. *IDL (Interface Definition Language) defined in The Common Object Request Broker*, volume Architecture and Specification, version 2.3.1. OMG Group, October 1999.

[55] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

[56] M. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[57] M. Parkinson and G. Bierman. Separation logic and abstraction. In *ACM SIGPLAN Notices*, volume 40, pages 247–258. ACM, 2005.

[58] D. J. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

[59] A. Raad. *Unconfirmed*. PhD thesis, Imperial College London, To appear.

[60] J. Reynolds. Precise, intuitionistic, and supported assertions in separation logic. *Slides from the invited talk given at the MFPS XXI Available at authors home page: http://www.cs.cmu.edu/ jcr*, 2005.

[61] J. C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM, 1978.

[62] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial perspectives in computer science*, pages 303–321, 2000.

[63] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[64] R. Russell, D. Quinlan, and C. Yeoh. Filesystem hierarchy standard. *V2*, 3:29, 2004.

[65] G. D. Smith. *Local reasoning about web programs.* PhD thesis, Imperial College London (University of London), 2011.

[66] C. Sophocleous. Masters thesis. *Imperial College London*, 2012.

[67] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. *ACM SIGMOD Record*, 30(2):413–424, 2001.

[68] P. Thiemann. A type safe DOM API. In *Proceedings DBPL, 169–183*, 2005.

[69] V. Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007.

[70] M. J. Wheelhouse. *Segment Logic.* PhD thesis, Imperial College London, 2012.

[71] L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, et al. Document Object Model (DOM) level 1 specification. *W3C Recommendation*, 1, 1998.

[72] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[73] World Wide Web Consortium. *HTML 4.01 Specification*, Dec. 1999.

[74] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *Computer Aided Verification*, pages 385–398. Springer, 2008.

[75] H. Yang and P. OHearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures*, pages 281–335. Springer, 2002.

[76] U. Zarfaty. *Context logic and tree update.* PhD thesis, Citeseer, 2007.

[77] U. Zarfaty and P. Gardner. Local reasoning about tree update. *Electronic Notes in Theoretical Computer Science*, 158:399–424, 2006.