Mechanizing Hoare Style Proof Outlines for Imperative Programs in Agda

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Olav Niels de Haas
born in Rotterdam, the Netherlands

To be defended on
July 11, 2022 at 9:00

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`https://pl.ewi.tudelft.nl`

Mechanizing Hoare Style Proof Outlines for Imperative Programs in Agda

Abstract

Formal verification of imperative programs can be carried out on paper by annotating programs to obtain an outline of a proof in the style of Hoare. This process has been mechanized by the introduction of Separation Logic and computer assisted verification tools. However, the tools fail to achieve the readability and convenience of manual paper proof outlines. This is a pity, because getting ideas and proofs across is essential for scientific research. I introduce a mechanization for proof outlines of imperative programs to interactively write human readable outlines in the dependently typed programming language and proof assistant Agda. I achieve this by introducing practical syntax and proof automation to write concise proof outlines for a simple imperative programming language based on λ-calculus. The proposed solution results in proof outlines that combine the readability of paper proof outlines and the precision of mechanization.

***Keywords*—** Programming Languages, Formal Verification, Separation Logic, Proof Automation, Proof Assistant
***Supplement Material*—** `https://github.com/Olavhaasie/hoare-proof-outlines`

Preface

I got the idea of a thesis in formal software verification from the course Software Verification given by Arjen. This course introduced me to the programming language Agda with which I was fascinated from the start. I hope this work makes formal verification of programs more accessible to other students and programmers alike. During my thesis I learned a lot about programming with dependent types, formal software verification and Agda. These will forever change the way I look at programming.

The final result of this thesis would not have been possible without the help of some incredible people. First and foremost I would like to thank Arjen for being my mentor and the weekly meetings we had even after he left the TU Delft. I would also like to thank my second supervisor Jesper, who taught me about the inner workings of Agda and provided useful feedback on implementation and writing. I also want to thank the late Eelco, who inspired me and many other students to study the field of programming languages through his engaging courses. I am also thankful for my friends Joris, Roel and Rutger for the enjoyable and productive thesis sessions on campus. Finally, I am grateful for the support of my friends Daan, Martijn and Laura.

Olav de Haas
Delft, the Netherlands
July 1, 2022

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The cost of poor software quality in 2020 was estimated to be $2.08 trillion in the US alone (Krasner 2021). This estimate is composed of multiple types of failures in software quality, but operational software failures are the largest contributor estimated to be $1.56 trillion in losses. This cost has increased by 22% over the last two years (Krasner 2018). Most codebases were found to use imperative programming languages, such as Java, C and C++ (Krasner 2021). Imperative languages use a sequence of statements with side effects to change the state of the program. In low-level imperative programming languages, failures often originate from unsafe use of pointers. An infamous example is Heartbleed, a major vulnerability in the openSSL cryptographic library, caused by buffer overflow, which is a consequence of unsafe pointer handling (Carvalho et al. 2014).

Software failures arise from mistakes in programs made by programmers. A way of mitigating mistakes is software testing. Programmers write tests that aim to find bugs in a program by running the program on some input and verifying if the result is equal to the expected result. However, tests only provide evidence that the program does not go wrong for a particular set of inputs. Tests are not an exhaustive proof that the program under test returns the correct result for every possible input or circumstance. There are always test cases a software tester has not thought of. A quote by Dijkstra sums it up well: "Testing shows the presence, not the absence of bugs" (Buxton and Randell 1970). This is why mistakes still slip through in programs and potentially cause a software failure when the program is run on unexpected input. This is especially true for imperative programming languages where the state space can become huge if not carefully designed. In order to show the absence of bugs, we turn to formal software verification.

Formal verification is the act of formally verifying properties of software. For example, we can verify that a program does not try to read a null pointer or that a program correctly sorts a list. We achieve these results by an exhaustive mathematical proof that a program follows a specification. In general, formal verification tries to answer the question: "Have we constructed a program that follows its specifications?". Consequently, given a correct specification of a program, formal verification shows the absence of bugs. Formal verification is an established method for proving correctness of programs as illustrated by CompCert (Leroy 2021), a verified correct C compiler, or the proof of correctness of the HMAC algorithm in openSSL (Beringer et al. 2015).

An approach to formal verification is deductive reasoning. We interpret programs as mathematical objects and use mathematics and logic to reason about programs. Hoare (1969) described a formal system for proving properties of programs. Based on the work of Dijkstra (1968) and Floyd (1967), Hoare also showed how to construct readable proofs of programs, known as proof outlines. A program is annotated by formal specifications of the program state at every step of the program to realize the outline of a proof. Proof outlines are an excellent way of introducing formal verification of programs to students and programmers. Proof outlines are still used in research to convey reproducible readable proofs to peers (O'Hearn 2019).

A long unsolved problem in formal verification of imperative programs was the handling of pointers. It was difficult to create an intuitive axiomatic treatment for the problem of pointer aliasing (Bornat 2000), where two seemingly different pointers point to the exact same location. Modifying the value at a location had the chance to change value that other pointers point to. It was not until the turn of the millennium that the joint work of O'Hearn, Reynolds, and Yang (2001) and Reynolds (2002) presented a breakthrough solution known as Separation Logic, an extension of the system of Hoare that allowed for reasoning about programs using separated state. Separation Logic elegantly solves the problem of pointer aliasing by building state from smaller separate pieces of state. Because of this separation, pointers are guaranteed to be separate locations and, therefore, not alias. Another reason why separation logic is so successful is by the introduction of the frame rule by O'Hearn, Reynolds, and Yang (2001) that makes the logic scalable (Pym, Spring, and O'Hearn 2019). Furthermore, the frame rule guarantees that programs do not touch any locations outside of their specification. Nowadays, Separation Logic is used as a basis for many formal program verification tools (O'Hearn 2019).

Reasoning about programs can quickly become complex and tedious, so we use proof assistants, such as Coq or Isabelle/HOL, to allow for interactive theorem proving. An implementation of a formal system of reasoning in a proof assistant is known as a mechanization. This method of verification is detailed and precise and, therefore, requires extensive knowledge on formal reasoning and program semantics. Currently, there exist mechanizations in the proof assistant Coq, such as Iris (Jung et al. 2018) and VST-Floyd (Cao et al. 2018), to interactively write proofs of program specifications by deductive reasoning based on Separation Logic. However, a readable proof in the style of Hoare is lost through complex automation and proof scripts. The steps of reasoning are not directly clear to a reader and an understanding of the framework is necessary in order to read the proofs.

In this thesis I present a mechanization of proof outlines that combine the readability of Hoare style proof outlines and the precision of mechanizations in proof assistants. I present an implementation of this mechanization in the dependently typed programming language Agda (Norell 2007; 2008; Agda Development Team 2021). Agda is a functional programming language based on Martin-Löf type theory (Martin-Löf 1982). Through the Curry-Howard correspondence, which shows the correspondence between propositions and types, we can write formal proofs as purely functional programs (Wadler 2015). So, Agda also functions as a proof assistant to interactively write proofs. This results in writing interactive proof outlines in the style of Hoare that are automatically verified by Agda.

1.1 Goal

Below follows the goal for this Master's thesis:

Write readable proof outlines for imperative programs in Agda.

1.2 Contributions

The contributions of this thesis can be summarized by the following:

1. We define a syntax for axioms and rules to write mechanized proof outlines for imperative programs in the style of Hoare and demonstrate an implementation in Agda (See Chapter 4).

2. We define proof automation by reflection to hide tedious proofs and demonstrate in Agda that this results in shorter and more concise proof outlines (See Chapter 5).
3. We show that the previous contributions produce a mechanization of readable Hoare style proof outlines in a case study on an imperative program that copies a list structure (See Chapter 6).

See Appendix A for a reference of the implementation in Agda.

Chapter 2

Separation Logic

Before we can define a mechanization for reasoning about an imperative programming language, we have to define a formal method of reasoning about mutable state. Separation Logic (SL) achieves this by separating state into disjoint shapes of state and only reasoning about the minimal state necessary in order to assert that the rest of the state remains unaffected. In this chapter we will explore and define the core specifications and how to combine them into more practical specifications.

2.1 High-Level Overview

The idea of SL builds on early work by Burstall (1972) and the logic of Bunched Implications (O'Hearn and Pym 1999). SL is a formal system for reasoning about programs that mutate data structures (O'Hearn, Reynolds, and Yang 2001; Reynolds 2002). SL is also an assertion language for describing state as separated parts. In this chapter we will focus on using SL for specifying mutable state of imperative programs. In Chapter 3 we will focus on defining the formal reasoning method.

In SL we reason about specifications of state, in contrast to exact instances of state. We call such a state a *store* or *heap* that contain locations that point to stored values, referring to the dynamically allocated heap in programming languages. The abstract notion of memory proves useful in formal reasoning, since we do not have to mention the details of the implementation of the underlying store and we can use the same line of reasoning for all instances of stores.

Specifications are built up from combining singleton stores. A singleton is specified by the points-to relation, written as $l \mapsto v$, where l is a location and v the value l points to. Singletons can be combined with the separating conjunction, written as $H_1 * H_2$, where H_1 and H_2 are store specifications. The separating conjunction $H_1 * H_2$ describes a store that can be *separated* into the two disjoint specifications H_1 and H_2. Larger stores can be described from these simple core specifications. For example, the specification $l \mapsto 3 * k \mapsto 5$ describes a store that contains two values 3 and 5 at locations l and k respectively. Following from the definition of the separating conjunction, we also have that l and k are two unique locations. Therefore, $l \mapsto v * l \mapsto w$ is not a valid specification: a store can never be separated into two equal locations.

The separation addresses the problem of pointer aliasing. Pointer aliasing occurs when two or more pointers map to the same location and, therefore, updating the value that one of the pointers map to also changes the values that other pointers map to. Furthermore, deallocating an aliased pointer results in dangling pointers, which are pointers that map to no valid value. Reading from dangling pointers can lead to memory safety violations. So, in SL we eliminate these issues by using the separating conjunction to assert that locations are unique and the points-to relation to assert that pointers map to a valid object.

How can we use Separation Logic to reason about imperative programs? We can formally reason about programs through symbolic execution (Dijkstra 1968; Hoare 1969). We start by writing down the program and the properties of the program that we would like to show. The idea is to annotate the program with the properties at each step of the execution. At the beginning of the program, we write down the specification of the state that we assume to result in a valid execution of the program. At the very end of the program, we write down the specification of the store that should be valid after the program has finished execution. We write down the state in SL in between the lines enclosed by curly braces. Then, we symbolically step through the program while updating our specification of the state after each step. To illustrate this process, take a program that copies a pointer:

```
$\{l \mapsto 2\}$
k := ref 0 ;
$\{l \mapsto 2 * k \mapsto 0\}$
k := ! l
$\{l \mapsto 2 * k \mapsto 2\}$
```

In this example, we use a made-up language where `ref` creates a new pointer from a value, assigns it by `:=` and reads from a pointer with `!` sequenced by the `;` operator. We start with our precondition that a location `l` exists. Then we create a new location `k` with value 0. The specification after this allocation reflects the updated state, which is now also separated by the freshly allocated location. Then, we update the value that `k` points to with the value that `l` points to and update the specification accordingly. Finally, we arrive at a specification with a separate location `k` that maps to the same value of 2. SL is unambiguous in its final specification, the program creates a new location `k` different from `l` and not just an aliased pointer. In Chapter 3 we will formalize this process of reasoning further.

We have shown in the example that the program copies a pointer when it is assumed that a single location exists. However, when this program is embedded in larger programs, then the state may not be exactly $l \mapsto 2$. This would mean we have to redo this proof for every possible state that $l \mapsto 2$ is embedded in. For this reason, SL has the *frame rule* to solve this problem. The frame rule states that any proof also holds within any given frame separated by $*$. So, with the frame rule we can derive a proof of this program which satisfies the state of $l \mapsto 2 * H$, where H is the frame.

The frame rule is also the key to *local reasoning*. We can interpret the frame rule as reasoning about a separated part of a store while other separated parts of the store remain unchanged. A program that adheres to a certain specification will never access locations outside of its specification.

The "output" of a step in a proof outline might not be in the exact shape for the "input" of the next step in the outline. Therefore, we should be able to *reshape* our state specifications into other specifications that are not the same but carry the same meaning. In Separation Logic this concept is also known as predicate entailment.

2.2 Predicate Entailment

A predicate entailment is a recipe for reshaping one store specification into another. For two predicates H_1 and H_2 we write down predicate entailment as $H_1 \vdash H_2$, which says that every store that satisfies H_1 also satisfies H_2.

Store specifications in proof outlines are rarely in the correct shape for the next step of a proof outline. So, we use predicate entailment to reshape our specifications in order to advance in our proof outline. Take the following snippet of a proof outline where we have arrived at a state of two locations `l` and `k` and our next step is to swap the values of the two locations:

```
{k ↦ 1 ∗ l ↦ 0}
?
{l ↦ 0 ∗ k ↦ 1}
swap l k
```

Swap takes a state in the shape of $l \mapsto v * k \mapsto w$ and outputs a state of shape $l \mapsto w * k \mapsto v$, but our state at the start of the outline is of shape $k \mapsto 1 * l \mapsto 0$. Now these shapes are not equal, so, we cannot execute our call to swap. The question mark symbolizes this gap in our reasoning. However, these two shapes clearly describe the same set of states. A state that separates l from k is the same as one that separates k from l. So, we use entailment to show that $k \mapsto 1 * l \mapsto 0$ follows from $l \mapsto 0 * k \mapsto 1$. In other words, we can reshape our state into one that fits the specification of the swap function. Now we can fill in our gap with this entailment and complete the outline. Finally, we arrive at the state where the values of l and k have been swapped.

```
{k ↦ 1 ∗ l ↦ 0}
{l ↦ 0 ∗ k ↦ 1}  # using entailment
swap l k
{l ↦ 1 ∗ k ↦ 0}
```

The previous example actually demonstrates one of the laws of $*$, namely commutativity. The separating conjunction is also associative, which means that the order of application of $*$ does not matter. Furthermore, there exists an identity predicate **emp**, which describes the empty state. The identity **emp** can be added to and removed from every separation, since any specification can be separated into itself and an empty state. We can generalize these laws in the following lemma.

Lemma 2.2.1 (Separating conjunction laws form a commutative monoid (**emp**, $*$)).

$$\begin{array}{rcll}
(H_1 * H_2) * H_3 & \dashv\vdash & H_1 * (H_2 * H_3) & \textsc{$*$-associativity} \\
H_1 * H_2 & \dashv\vdash & H_2 * H_1 & \textsc{$*$-commutativity} \\
\mathbf{emp} * H & \dashv\vdash & H & \textsc{$*$-left-identity} \\
H * \mathbf{emp} & \dashv\vdash & H & \textsc{$*$-right-identity}
\end{array}$$

We use bi-entailment $\dashv\vdash$ as a shorthand for entailment in two directions. The bi-entailment $H_1 \dashv\vdash H_2$ says that any state that satisfies H_1 also satisfies H_2 and the other way around. Bi-entailment also defines an equivalence on predicates that is reflexive, symmetric and transitive.

Lemma 2.2.2 ($\dashv\vdash$ is an equivalence).

$$\dfrac{}{H \dashv\vdash H}\,\textsc{$\dashv\vdash$-reflexive} \qquad \dfrac{H_1 \dashv\vdash H_2}{H_2 \dashv\vdash H_1}\,\textsc{$\dashv\vdash$-symmetric} \qquad \dfrac{H_1 \dashv\vdash H_2 \qquad H_2 \dashv\vdash H_3}{H_1 \dashv\vdash H_3}\,\textsc{$\dashv\vdash$-transitive}$$

The laws of $*$ show how we can reshape a full specification in SL. However, we cannot yet reason about shapes separately. We would like to reason with the minimal state necessary. This not only results in a simpler specification, but it also assures that we do not access or mutate any other resources outside of our specification. The previous outline for swap is an excellent example of this. The swap function only takes the minimal state necessary: two locations l and k. However, we have a problem when we have a state that contains more than just two locations, because swap is only defined for two locations. So, we need a rule to only reshape a separate part without changing other separate parts. In SL this is formalized as the monotonicity of the separating conjunction.

Lemma 2.2.3 (Monotonicity of separating conjunction with respect to entailment)**.**

$$\textsc{$*$-left-mono}\quad \frac{H_2 \vdash H_2'}{H_1 * H_2 \vdash H_1 * H_2'} \qquad \textsc{$*$-right-mono}\quad \frac{H_1 \vdash H_1'}{H_1 * H_2 \vdash H_1' * H_2}$$

$$\textsc{$*$-mono}\quad \frac{H_1 \vdash H_1' \qquad H_2 \vdash H_2'}{H_1 * H_2 \vdash H_1' * H_2'}$$

Monotonicity allows us to reshape a separate part of a specification while leaving the other part unchanged. This is formalized as left monotonicity, where the left part of a separating conjunction remains unchanged, and right monotonicity, where the right part of a separating conjunction remains unchanged. We can also combine them into general monotonicity, where we reshape both parts of a separation.

In order to see how we can use these rules in proof outlines, we take the swap example again, but now we start in a state that contains one more location:

```
{(l ↦ 0 ∗ k ↦ 1) ∗ m ↦ 2}
  {l ↦ 0 ∗ k ↦ 1}          # Right monotonicity
  swap l k
  {l ↦ 1 ∗ k ↦ 0}
{(l ↦ 1 ∗ k ↦ 0) ∗ m ↦ 2}  # Restore
```

We apply right monotonicity to temporarily forget the separated $m \mapsto 2$. Then, the indented part proceeds the line of reasoning but now with only the minimal state necessary. We swap the values and we restore the part that we had temporarily forgotten about. We know for sure that the location m remains unchanged, because it is not included in the actual swapping. This also introduces the frame rule, which we will formally define in Chapter 3.

2.3 Store Predicates

So far, we have only seen predicates that describe state of separated locations and the empty state, using the separating conjunction $*$, points-to relation $\mapsto$ and empty predicate **emp**. However, with just these predicates we cannot describe more complex states. What if we want to, for example, describe a state that does not contain an odd number? This is simply not possible. So, usually in SL we define more predicates and combine them into more complex specifications. In Table 2.1 we list the predicates we will be using and the sets of stores describe.

Most notable and useful is the pure predicate. With a pure predicate we can embed statements from propositional logic into our specifications. They are called *pure* because they are *independent* of the store. This will be very useful in our mechanization, since we can say something about the locations and values in our specification. We define pure predicates to specify the empty store, since we can easily combine the empty store with the separating conjunction using the left and right identity of **emp**. For example, we can now describe a state that contains a single value that is greater than 10 with $[\![v > 10]\!] * l \mapsto v$, which can be read as: the empty store and v is greater than 10 and separately a location l that maps to v. For convenience, we define some rules for reasoning with pure predicates that introduce, eliminate and map a proposition in a separation.

Lemma 2.3.1 (Rules for pure predicates)**.**

$$\textsc{pure-intro}\quad \frac{P}{H \vdash [\![P]\!] * H} \qquad \textsc{pure-elim}\quad \frac{}{[\![P]\!] * H \vdash H} \qquad \textsc{pure-map}\quad \frac{P \Rightarrow Q}{[\![P]\!] \vdash [\![Q]\!]}$$

Name	Notation	Set of Stores
Separating conjunction	$H_1 * H_2$	Stores that can be separated into two disjoint stores such that one satisfies H_1 and the other satisfies H_2.
Points-to relation	$l \mapsto v$	Stores that contain exactly one location l that maps to the value v.
Empty predicate	**emp**	Stores that are empty.
Pure predicate	$[\![P]\!]$	Set of stores that are empty and where the proposition P holds independent from the store.
Top	$\top$	All stores.
Bottom	$\perp$	No store.
Conjunction	$H_1 \land H_2$	Stores that are described by H_1 and H_2.
Disjunction	$H_1 \lor H_2$	Stores that are either described by H_1 or H_2.
Complement	$\neg H$	Stores that are not described by H.
Existential quantifier	$\exists x.\ H$	Stores described by H for which there exists an x.
Universal quantifier	$\forall x.\ H$	Stores that are described by H for any x.

Table 2.1: List of predicates in separation Logic and the sets of stores they describe.

Then, we define top and bottom in SL. We can interpret them as *true* and *false*. $\top$ is true for every store, whereas $\perp$ is false for every store. We use $\top$ in a separation to specify that a part is contained in a store. For example, $l \mapsto v * \top$ reads as a location l that maps to v and separately any store. So, any store that contains at least one location satisfies this specification. On the contrary, using $\perp$ in a separation is not very useful, since that would always imply no store. Instead, we use $\perp$ to define contradictions. Remember that $*$ separates two disjoint parts, so, whenever we have a state that separates two equal locations, we should get a contradiction. In other words, an equal separation entails false. Now we can fully define this statement in SL using entailment.

Lemma 2.3.2 (Single conflict).

$$l \mapsto v * l \mapsto w \vdash \perp$$

We also define the logical connectives and quantifiers from first order logic in SL. The conjunction combines two specifications and states that they must both be satisfied for the same store. This is very different from the separating conjunction, which states that both specifications must be satisfied for *separate* parts of the store. So, $l \mapsto v * k \mapsto w$ describes a store that contains two locations, whereas $l \mapsto v \land k \mapsto w$ describes a store that contains one location, where l and k must be the same location. The disjunction describes a store that is specified by either of the specifications. We can use this operator to define data types that have alternatives, such as an optional location: **emp** $\lor\ l \mapsto v$, a store that either contains no location or one location. The existential quantifier describes a store that is satisfied by a specification for some value of any type. The variable that represents this value can be referenced in the specification. For example, the specification $\exists v.\ l \mapsto v$ describes a store

$l \mapsto v \land \mathbf{emp}$	The set of stores that is empty and contains a single location. There exists no store that is empty and not empty, so this statement describes the empty set of stores.
$[\![v \neq 2]\!] * l \mapsto v * \top$	The set of stores that *contains* a location l that points to a value v that is not equal to 2. The set of stores separated by $\top$ can be any set of stores as long as the store does not contain the location l.
$(\mathbf{emp} * l \mapsto v) * \neg(l \mapsto v * \mathbf{emp})$	A store described by this statement must be separated into a location l that points to value v and a store that does not contain the location l. The **emp** can be left out, since any store can be separated into the empty store and something else. So, this statement describes the set of stores that contain at least the location l.
$\forall l.\ \exists v.\ l \mapsto 2v$	The universal quantifier does not specify that every location in the store must be an even number, but rather that the location of the store may be any location. Actually, the universal quantifier for l is not necessary here, since we already implicitly universally quantify over the variables that we use. The existential quantifier says that there exists some number v such that any value l points to twice v. The store that this specification describes must also have exactly one location. So, it describes the set of stores that consist of a single location l that points to an even number.

Table 2.2: Examples of statements in Separation Logic and the sets they describe.

that has one location with some value v. The universal quantifier can be used in the same way, but any possible value of v satisfies the specification. Finally, we define laws to reshape separations of quantifiers.

Lemma 2.3.3 (Rules for separating conjunction and quantifiers)**.**

$$
\begin{array}{lcllr}
(\exists x.\ H_1) * H_2 & \dashv\vdash & \exists x.\ (H_1 * H_2) & (\text{if } x \text{ is } \textbf{not} \text{ free in } H_2) & *\text{-EXISTS} \\
(\forall x.\ H_1) * H_2 & \vdash & \forall x.\ (H_1 * H_2) & (\text{if } x \text{ is } \textbf{not} \text{ free in } H_2) & *\text{-FORALL}
\end{array}
$$

For more examples of specifications see Table 2.2.

2.4 Definitions

Now that we have seen how to combine store specifications and interpret them, we will move on to finding correct definitions that reflect their meaning. Remember that we are creating a mechanization of proof outlines in Agda, so, we will define the operators in the type theory of Agda. In this section we will follow the definitions from Charguéraud (2020).

Store specifications in SL define some proposition about stores. So, specifications are predicates over stores, or, as we already seen, sets of stores. We define the type of a store predicate as follows:

Definition 2.4.1 (Store predicate)**.** A store predicate has type *Pred* : *Store* → *Set*.

The *Set* type is the type of types, which corresponds to a any logic proposition. Furthermore, we write definitions using $A \triangleq B$, which means "A is defined as B".

2.4.1 Entailment and Equivalence

Similar to how we started this chapter, we will begin by defining entailment and equivalence of store predicates.

Definition 2.4.2 (Store predicate entailment)**.** A store predicate H_1 entails another store predicate H_2, written $H_1 \vdash H_2$, if every store σ that satisfies H_1 also satisfies H_2.

$$H_1 \vdash H_2 \quad \triangleq \quad \forall \sigma.\ H_1\ \sigma \Rightarrow H_2\ \sigma$$

Definition 2.4.3 (Store predicate bi-entailment)**.** A store predicate H_1 is equivalent to another predicate H_2, written $H_1 \dashv\vdash H_2$, when H_1 entails H_2 and H_2 entails H_1.

$$H_1 \dashv\vdash H_2 \quad \triangleq \quad (H_1 \vdash H_2) \wedge (H_2 \vdash H_1)$$

In the definition of entailment, we apply the store of type *Store* to the predicates of type *Pred*, to get a *Set* type, therefore, entailment is not a predicate but a proposition. The thick right arrow ($\Rightarrow$) is implication from the host logic, which corresponds to the function type according to the Curry-Howard correspondence. The definition reads as: every store σ that satisfies H_1 also satisfies H_2. Bi-entailment is simply defined in terms of entailment in both directions. From these definitions it becomes clear that $\dashv\vdash$ is an equivalence (see Lemma 2.2.2).

2.4.2 Separation Algebra

Now we move on to predicates that specify what a store contains. These include the separating conjunction ($*$), points-to relation ($\mapsto$) and empty predicate (**emp**). We will define these predicates in a low-level separation algebra that relates stores directly. This creates an generic interface for stores to implement and from this minimal algebra we can derive more complex predicates without having to implement all the predicates for every type of store. The predicates in SL can be seen as a high-level composable method of writing specifications defined in the less usable low-level algebra.

The separation algebra has four components. The first is what it means to separate a store. We define this separation by a ternary relation on stores that relates two disjoint stores to a store that is the union of the stores. This relation is *ternary*, because it relates *three* objects. We write this relation as $\sigma_1 \uplus \sigma_2 \doteq \sigma$, which says that σ can be separated into two disjoint stores σ_1 and σ_2. The ternary relation is not defined for two overlapping stores σ_1 and σ_2. Besides the relation, we need two specific instances of stores, namely a singleton store and an empty store instance. In essence, we require a store to have an empty instance, singleton instance and a relation to combine these two into larger stores. The last component is an equivalence on stores that we use to relate stores to each other. The ternary relation and predicates should respect the equivalence on stores. We formally define our separation algebra as:

Definition 2.4.4 (Separation algebra)**.** A store has a separation algebra defined by the following components:

1. A disjoint ternary relation: $\sigma_1 \uplus \sigma_2 \doteq \sigma$, where the union of two disjoint stores σ_1 and σ_2 is equivalent to σ.

2. An equivalence on stores: $\sigma_1 \simeq \sigma_2$.

3. An empty store instance: $\varnothing$.
4. A singleton store instance: $\{l \to v\}$, where the location l maps to a value v.

Now we can define the separating conjunction, points-to relation and the empty predicate in our new separation algebra.

Definition 2.4.5 (Separating conjunction)**.**

$$H_1 * H_2 \quad \triangleq \quad \lambda\sigma.\ \exists\sigma_1\exists\sigma_2.\ (\sigma_1 \uplus \sigma_2 \doteq \sigma) \wedge H_1\ \sigma_1 \wedge H_2\ \sigma_2$$

Definition 2.4.6 (Points-to relation)**.** The points-to relation, written $l \mapsto v$, asserts a store contains a single location l that maps to value v.

$$l \mapsto v \quad \triangleq \quad \lambda\sigma.\ \sigma \simeq \{l \to v\}$$

Definition 2.4.7 (Empty predicate)**.** The empty predicate, written **emp**, asserts a store is empty.

$$\mathbf{emp} \quad \triangleq \quad \lambda\sigma.\ \sigma \simeq \varnothing$$

We define the separating conjunction by existentially quantifying over two stores that separate the store into two disjoint stores and satisfy the two predicates. The definition reads as follows: a store satisfies the conjunction if the store can be separated into two disjoint stores where one satisfies the first and the other the second specification. This definition is also suited to show monotonicity of the separating conjunction (see Lemma 2.2.3) with respect to entailment. However, in order to show the other laws of separating conjunction (see Lemma 2.2.1) we also require that the disjoint ternary relation is associative, commutative and $\varnothing$ is the identity. We express these laws formally as follows:

Lemma 2.4.8 (Separation algebra laws)**.**

$$\begin{array}{rcll}
\sigma_1 \uplus \sigma_{23} \doteq \sigma_{123} \wedge \sigma_2 \uplus \sigma_3 \doteq \sigma_{23} & \Rightarrow & \exists\sigma_{12}.\ \sigma_1 \uplus \sigma_2 \doteq \sigma_{12} \wedge \sigma_{12} \uplus \sigma_3 \doteq \sigma_{123} & \uplus\text{-}\textsc{left-associativity} \\
\sigma_1 \uplus \sigma_2 \doteq \sigma_{12} \wedge \sigma_{12} \uplus \sigma_3 \doteq \sigma_{123} & \Rightarrow & \exists\sigma_{23}.\ \sigma_1 \uplus \sigma_{23} \doteq \sigma_{123} \wedge \sigma_2 \uplus \sigma_3 \doteq \sigma_{23} & \uplus\text{-}\textsc{right-associativity} \\
\sigma_1 \uplus \sigma_2 \doteq \sigma & \Leftrightarrow & \sigma_2 \uplus \sigma_1 \doteq \sigma & \uplus\text{-}\textsc{commutativity} \\
\varnothing \uplus \sigma_1 \doteq \sigma_2 & \Leftrightarrow & \sigma_1 \simeq \sigma_2 & \uplus\text{-}\textsc{left-identity} \\
\sigma_1 \uplus \varnothing \doteq \sigma_2 & \Leftrightarrow & \sigma_1 \simeq \sigma_2 & \uplus\text{-}\textsc{right-identity}
\end{array}$$

2.4.3 Other Definitions

Finally, we give definitions for the remaining predicates.

Definition 2.4.9 (Pure predicate)**.** The pure predicate, written $[\![P]\!]$, asserts a proposition P and the empty store.

$$[\![P]\!] \quad \triangleq \quad \lambda\sigma.\ \sigma \simeq \varnothing \wedge P$$

Definition 2.4.10 (Logical connectives and constants)**.**

$$\begin{array}{rcl}
\top & \triangleq & \lambda\sigma.\ \text{true} \\
\bot & \triangleq & \lambda\sigma.\ \text{false} \\
H_1 \wedge H_2 & \triangleq & \lambda\sigma.\ H_1\ \sigma \wedge H_2\ \sigma \\
H_1 \vee H_2 & \triangleq & \lambda\sigma.\ H_1\ \sigma \vee H_2\ \sigma \\
\neg H & \triangleq & \lambda\sigma.\ \neg(H\ \sigma)
\end{array}$$

Definition 2.4.11 (Quantifier predicates)**.** Quantifier predicates existentially and universally quantify over variables in a store predicate H.

$$\begin{array}{rcl}
\exists x.\ H & \triangleq & \lambda\sigma.\ \exists x.\ H\ \sigma \\
\forall x.\ H & \triangleq & \lambda\sigma.\ \forall x.\ H\ \sigma
\end{array}$$

Chapter 3

Hoare Logic

In the previous chapter we have seen an intuition of proof outlines and how to use Separation Logic to describe state. In this chapter we will define a formal system for reasoning about programs by combining Separation Logic with Hoare Logic. This formal system will lay a foundation for writing mechanized proof outlines for imperative programs.

3.1 Hoare Triples

We define the properties of a program using Hoare triples (Hoare 1969). A triple consists of a precondition, a program and a postcondition that we write down as follows:

$$\{H\}\, t\, \{Q\}$$

where H is the condition that should be satisfied before execution of program t and Q the condition that is satisfied after the execution of the program. We express the conditions using Separation Logic. H describes the state at the start of the program and Q the state at the end of the program. So, given a state of shape H then the program t will successfully execute and terminate to a state of shape Q. Furthermore, we can mention the return value of the program in the postcondition to assert some proposition about what the program evaluates to. We can concisely write down the specification of a program in a Hoare triple. For example, a function `add` that adds the values at two locations together can be specified by the following triple:

$$\{l \mapsto n * k \mapsto m\}\, \texttt{add l k}\, \{w.\ [\![w = n + m]\!] * l \mapsto n * k \mapsto m\}$$

Assuming that two locations that map to numbers exist, the program `add l k` evaluates to a value that is equal to the sum of the numbers at the two locations and does not alter the state in any further way. We bind the return value of the program to the variable w and add a pure predicate that states that the program does in fact return a sum. Then, given a definition of `add`, we should be able to prove that the triple is satisfied by a proof outline.

In the program we reference the locations that correspond to locations in the store as l and k. However, in the body of `add` function, the values are referenced by their argument names. So, we need a way to transfer the information that the arguments correspond to the locations. We could define a substitution algorithm to substitute the arguments with the locations, but this changes the structure of the program we are reasoning about and it becomes difficult to reason about programs that have free variables. Instead, we treat variables as a resource by extending our notion of Hoare triples with an environment (Parkinson, Bornat, and Calcagno 2006). We lookup the values that variables map to as we write our outline. We express triples extended with an environment as:

$$\{\eta \vdash H\}\, t\, \{Q\}$$

where η is an environment that maps variables to values. The meaning of this triple extended with an environment becomes: given an environment η and a state of shape H then the program t will successfully execute and terminate to a state of shape Q. The difference between an environment and a store is that values in the environment are immutable and not persistent throughout a program, whereas the values in the store are mutable and are persistent in a program. For example, binding a variable in a function body does not make it accessible outside of the function body after application of the function. Once a variable is bound no other value can be assigned to it. Whereas a value allocated in the store in a function body will still exist after the application of the function, given that the location was not deallocated. This extension of Hoare triples allows us to define program specifications for programs with variables and easily specify what the variables should represent. Now we can specify a triple for the body of `add`, where the arguments are named x and y as:

$$\{\{x \to l, y \to k\} \vdash l \mapsto n * k \mapsto m\}\, \textit{add-body}\, \{w.\ [\![w = n + m]\!] * l \mapsto n * k \mapsto m\}$$

We write down the environment as a map that maps the variables to their corresponding values. For the *add-body* to evaluate correctly, there have to be two variables that represent two locations on the heap.

Ultimately, our goal is to write proof outlines for Hoare triples. Proof outlines can be seen as a recipe to construct a Hoare triple and, therefore, verify their correctness. Until now we have seen proof outlines without strict rules on how to construct them. However, a mechanization requires a precise definition of Hoare triples and inference rules, independent of the proof assistant used, which in our case is Agda. Therefore, we will define Hoare triples extended with Separation Logic as a precise formal system. As any formal system, we construct Hoare triples with inference rules to derive new triples from axioms. The inference rules are split into two types: structural rules and language construct rules. The structural rules are valid for any programming language, whereas the language rules are specific to constructs of the programming language. An example of a language construct rule would be to derive a triple for an if-then-else branching construct. The axioms are triples of the core expressions of the programming language, for example, assigning a value to a location. Since a program in any programming language is just a combination of the core constructs of the language, we can use the inference rules to construct Hoare triples for larger programs. In the remainder of this chapter we will define the structural rules for Hoare triples and see how they can be used to write proof outlines independent of a subject programming language.

3.2 Structural Rules

The structural rules for Hoare triples are inference rules that can be used to derive new triples. These rules can be used for triples of any programming language: they are structural to triples. In this section we will define the structural rules that are necessary for writing proof outlines for imperative programs.

3.2.1 Rule of Consequence

The first rule is the rule of consequence introduced by Hoare (1969) in his paper on triples. If we have a program that satisfies a precondition H, then the program is also satisfied by every precondition that entails H. Furthermore, if a program satisfies a postcondition Q, then the program also satisfies any postcondition that is entailed by Q. The consequence rule allows us to reshape a precondition and a postcondition by entailment. We separate the inference rule into pre- and post-consequence, which can be used to derive the rule of consequence for both pre- and postconditions. We express the inference rules formally as:

Lemma 3.2.1 (Rule of consequence)**.**

$$\textsc{pre-consequence}\quad \frac{H \vdash H' \qquad \{\eta \vdash H'\}\, t\, \{Q\}}{\{\eta \vdash H\}\, t\, \{Q\}} \qquad\qquad \textsc{post-consequence}\quad \frac{\{\eta \vdash H\}\, t\, \{Q'\} \qquad Q' \mathrel{\dot{\vdash}} Q}{\{\eta \vdash H\}\, t\, \{Q\}}$$

$$\textsc{consequence}\quad \frac{H \vdash H' \qquad \{\eta \vdash H'\}\, t\, \{Q'\} \qquad Q' \mathrel{\dot{\vdash}} Q}{\{\eta \vdash H\}\, t\, \{Q\}}$$

Since postconditions are predicates over a store and a return value, we define a version of entailment that works for postconditions. A postcondition Q_1 entails Q_2, if every possible return value and store that satisfy Q_1 also satisfy Q_2. We define this version of entailment using the operator $\dot{\vdash}$, which is usual operator for entailment with a dot above. The same laws of entailment also apply to postcondition entailment. Postcondition entailment can be expressed in terms of normal predicate entailment:

Definition 3.2.2 (Postcondition entailment)**.**

$$Q_1 \mathrel{\dot{\vdash}} Q_2 \quad \triangleq \quad \forall v.\ (Q_1\ v) \vdash (Q_2\ v)$$

We can now create a derivation of a triple using the rule of consequence. For example, we can show a triple for alternative conditions than their original definition:

$$\frac{\dfrac{}{\mathbf{emp} * l \mapsto v \vdash l \mapsto v} \qquad \{l \mapsto v\}\, \texttt{dref}\ l\, \{k \mapsto l * l \mapsto v\} \qquad \dfrac{}{k \mapsto l * l \mapsto v \vdash l \mapsto v * k \mapsto l}}{\{\mathbf{emp} * l \mapsto v\}\, \texttt{dref}\ l\, \{l \mapsto v * k \mapsto l\}}$$

Here we derive a new triple for `dref` l using the rule of consequence in a tree derivation. The definition of `dref` that we might have defined is on top of the rule, whereas the new triple we would like to have is at the bottom of the rule. The function takes a location and allocates a new location that maps to the given one. We leave out the environment and return value for simplicity. Left of the triple is the entailment that entails the precondition by left identity of **emp** and to the right is the entailment of the postcondition by commutativity of $*$. We can also write the derivation in a linear style to get a proof outline:

```
{emp ∗ l ↦ v}
{l ↦ v}          # by left identity of emp
dref l
{k ↦ l ∗ l ↦ v}
{l ↦ v ∗ k ↦ l}  # by commutativity of ∗
```

A proof outline is just a more readable version of a tree derivation. We name the rules that we apply in comments on the line that they are applied. The outline has a more natural flow to it, since you can read it from top to bottom.

The separation of the rule of consequence into a pre and post version is useful for proof outlines. If we only used the complete rule of consequence, we have to match the number of entailments for the pre- and postcondition. So, if we want to use more steps to reshape the precondition we have to give the identity entailment for postcondition. We can now use more steps to provide intermediate states of a proof for clarity without having to enter identity consequences to balance the rule of consequence. If we separate the rule of consequence, we can asymmetrically reshape the pre- and postcondition.

3.2.2 Frame Rule

The frame rule defines the notion of local reasoning. The rule was introduced by Ishtiaq and O'Hearn (2001) after Reynold's first paper on Separation Logic. If a program satisfies a condition, then that program also satisfies the condition separated by a frame. We can specify triples for a minimal state, which only include the necessary locations, and later embed it into any state. We can be certain that the frame is not mutated, since it is not included in the original triple. Therefore, the frame is the same for the pre- and postcondition. We define the frame rule formally as:

Lemma 3.2.3 (Frame rule).

$$\textsc{frame}\quad \frac{\{\eta \vdash H\}\, t\, \{Q\}}{\{\eta \vdash H * H'\}\, t\, \{Q \mathbin{*\cdot} H'\}}$$

Since Q is a postcondition and the frame H' a store predicate, we define a new heterogeneous separating conjunction which separates a postcondition and a store predicate, written as $*\cdot$, where the dot represents that the store predicate on the right hand side is quantified over a return value. We define the postcondition separating conjunction in terms of the separating conjunction:

Definition 3.2.4 (Postcondition separating conjunction).

$$Q \mathbin{*\cdot} H \quad \triangleq \quad \lambda v.\, (Q\ v) * H$$

Similar to the rule of consequence we can create a derivation tree of a triple. Let us take the `dref` example again, but now we frame the triple with a location:

$$\frac{\{l \mapsto v\}\, \texttt{dref}\, l\, \{k \mapsto l * l \mapsto v\}}{\{l \mapsto v * m \mapsto w\}\, \texttt{dref}\, l\, \{(k \mapsto l * l \mapsto v) * m \mapsto w\}}$$

We leave out the return value of the program, since we are not interested in it at the moment. Therefore, we can specify the postcondition as a store predicate. We implicitly bind the return value to nothing. Again, we can also write the derivation as a more readable proof outline:

```
{l ↦ v ∗ m ↦ w}
{l ↦ v}                  # frame with m ↦ w
dref l
{k ↦ l ∗ l ↦ v}
{(k ↦ l ∗ l ↦ v) ∗ m ↦ w}  # restore frame
```

3.2.3 Combined Consequence-Frame Rule

The previous two rules are used in proof outlines to reshape pre- and postconditions and forget a frame temporarily to reason with minimal state. However, when using the frame rule, the conditions are rarely of shape where the desired minimal shape is separated by the frame. Before applying the frame rule we often apply the rule of consequence first to get the desired shape. Therefore, we combine the rule of consequence and the frame rule into the consequence-frame rule.

We define the combined rule as follows. If we have a program that satisfies the precondition H and postcondition Q, then the program also satisfies every precondition that entails H and separately a frame and every postcondition that is entailed by Q and separately the same frame. We express the combined consequence-frame rule formally as:

Lemma 3.2.5 (Combined consequence-frame rule).

$$\textsc{consequence-frame}\quad \frac{H \vdash H_1 * H_2 \qquad \{\eta \vdash H_1\}\, t\, \{Q_1\} \qquad Q_1 \mathbin{*\cdot} H_2 \mathrel{\dot{\vdash}} Q}{\{\eta \vdash H\}\, t\, \{Q\}}$$

We do not separate the combined consequence-frame rule into a pre and post version, because we want to make it visually explicit in the outline where we reason with a smaller state. It should be clear to the reader where we temporarily leave a frame out and where we restore the frame again.

To illustrate how the rule creates a shorter outline, take the following triple of `dref`:

$$\{m \mapsto w * l \mapsto v\}\, \texttt{dref}\, l\, \{k \mapsto l * (l \mapsto v * m \mapsto w)\}$$

We want to derive this triple from the following triple:

$$\{l \mapsto v\}\, \texttt{dref}\, l\, \{k \mapsto l * l \mapsto v\}$$

We first have to reshape the precondition into correct shape before we can apply the frame rule. Without using the combined consequence-frame rule, the outline for this derivation looks like this:

```
{m ↦ w * l ↦ v}
{l ↦ v * m ↦ w}            # pre consequence by commutativity of *
{l ↦ v}                    # frame with m ↦ w
dref l
{k ↦ l * l ↦ v}
{(k ↦ l * l ↦ v) * m ↦ w}  # restore frame
{k ↦ l * (l ↦ v * m ↦ w)}  # post consequence by associativity of *
```

However, if we use the combined consequence-frame rule, we can combine the steps on line 2 and 3 and the steps on line 6 and 7. The outline would then look like this:

```
{m ↦ w * l ↦ v}
{l ↦ v}                    # frame m ↦ w by commutativity of *
dref l
{k ↦ l * l ↦ v}
{k ↦ l * (l ↦ v * m ↦ w)}  # restore frame by associativity of *
```

3.2.4 Embedding Rules

Finally, we define derivation rules to embed terms of our meta theory into SL. The first is a rule to embed a proposition into a precondition. If we have a triple that is satisfied if some proposition holds, then we also have a triple where that proposition is a pure predicate in the precondition. The second rule is used to embed variables that we universally quantify over in our meta theory into a precondition with an existential quantifier. If we have a triple that is satisfied for every x, then we can derive a triple where there exists an x such that the precondition of the triple is satisfied. We formally express these inference rules as follows:

Lemma 3.2.6 (Structural rules of triples).

$$\textsc{pure}\quad \frac{P \Rightarrow \{\eta \vdash H\}\, t\, \{Q\}}{\{\eta \vdash \llbracket P \rrbracket * H\}\, t\, \{Q\}} \qquad\qquad \textsc{exists}\quad \frac{\forall x.\ \{\eta \vdash H\}\, t\, \{Q\}}{\{\eta \vdash \exists x.\ H\}\, t\, \{Q\}}$$

3.3 Definition of Hoare Triples

Now that we have defined the inference rules for triples, we need a suitable definition of Hoare triples that adhere to these rules. A triple is satisfied if an environment and a state that satisfies the precondition make a program correctly evaluate to a state that satisfies the postcondition.

In order to create this definition, we need a notion of program evaluation. We define this as a relation on program terms, environment and state, which we write down as:

$$(t, \eta, \sigma \Downarrow v, \sigma')$$

The relation means that a program t executes correctly in environment η and state σ to a value v and state σ'. This relation is also known as the semantics of a programming language.

Now we can define a total correctness Hoare triple by the following definition:

Definition 3.3.1 (Total correctness Hoare triple).

$$^{\text{HOARE}}\{\eta \vdash H\}\, t\, \{Q\} \quad \triangleq \quad \forall \eta.\, \forall \sigma.\; H\; \sigma \Rightarrow \exists v.\, \exists \sigma'.\; (t, \eta, \sigma \Downarrow v, \sigma') \wedge (Q\; v\; \sigma')$$

Total correctness means that the program does not error or loop indefinitely. So, this definition is only used to reason about correct and terminating programs. A Hoare triple specifies evaluation in a whole state, whereas we want to use SL to reason about fragments of state. Therefore, we create a definition of Separation Logic triples where for every frame of shape H' a program satisfies the Hoare triple $^{\text{HOARE}}\{\eta \vdash H * H'\}\, t\, \{Q \mathbin{*\cdot} H'\}$. This essentially "bakes in" the frame rule (Birkedal, Torp-Smith, and Yang 2005; 2006). We achieve this by quantifying over a frame (Charguéraud 2020):

Definition 3.3.2 (Total correctness Separation Logic triple).

$$\{\eta \vdash H\}\, t\, \{Q\} \quad \triangleq \quad \forall H'.\; ^{\text{HOARE}}\{\eta \vdash H * H'\}\, t\, \{Q \mathbin{*\cdot} H'\}$$

Chapter 4

Hoare Style Reasoning

In the previous chapters we have defined a formal system for reasoning about programs. However, the programs have been abstract and independent of any actual imperative programming language. In order to write proof outlines for a concrete program, we require an imperative programming language. In this chapter we will define such a language and design Hoare triples to write mechanized Hoare style proof outlines for this specific language.

4.1 Imperative Programming Language

We will define an imperative language based on λ-calculus. The language will have the usual function abstraction and application that define a λ-calculus. In order to make it imperative, we will add a sequence operator and add operators to allocate, deallocate, read and write to locations on the heap. Furthermore, to keep our language simple and only focus on defining triples, the language will not be typed. We also want to easily encode algebraic data types such as lists and trees. So, we will also extend the language with sums and products to represent algebraic data as sums of products[1].

First, we will define a syntax that covers these features. Then, we will define semantics to show how the language should be evaluated. Finally, from the semantics we can derive suitable Hoare triples that specify the properties of every construct in the language.

By deriving the Hoare triples from the semantics of the language, we can easily define triples for a language that already has semantics.

4.1.1 Syntax

The language is divided into values v, variables x and terms t. The values serve two purposes: terms evaluate to values and values are stored on the heap. Variables are represented by names that are not keywords. Programs are constructed with terms. Since this is a language with a heap, we can also reference locations l on the heap as values. We express the syntax as a context-free grammar in Backus-Naur Form:

[1]See for example in Haskell: `https://wiki.haskell.org/Algebraic_data_type`

Definition 4.1.1 (Syntax of the language)**.** BNF grammar of the imperative language based on λ-calculus.

v	$::=$	`()`	unit
	$\mid$	l	location
	$\mid$	`(` v `,` v `)`	product
	$\mid$	**`left`** v $\mid$ **`right`** v	sum
	$\mid$	**`clos`** η x `=>` t	closure
	$\mid$	**`fix`** η f x `=>` t	recursive closure
t	$::=$	x	variables
	$\mid$	v	values
	$\mid$	`(` t `)`	parentheses
	$\mid$	`(` t `,` t `)`	create product
	$\mid$	**`fst`** t $\mid$ **`snd`** t	deconstruct product
	$\mid$	**`left`** t $\mid$ **`right`** t	create sum
	$\mid$	**`switch`** t `|`**`left`** x `=>` t `|`**`right`** x `=>` t	deconstruct sum
	$\mid$	t `;` t	sequence
	$\mid$	$t\ t$	function application
	$\mid$	**`fun`** x `=>` t	function
	$\mid$	**`rec`** f x `=>` t	recursive function
	$\mid$	**`let`** x `=` t **`in`** t	let binding
	$\mid$	**`ref`** t	allocate
	$\mid$	**`free`** t	deallocate
	$\mid$	`!` t	dereference
	$\mid$	t `:=` t	assign

The syntax makes a distinction between non-recursive functions **`fun`** and recursive functions **`rec`**. The difference being that a recursive function also gets itself as an argument named f to be recursively called inside the body, since the functions are not named. Multi argument functions can be constructed by currying functions. Functions also have a value counterpart: the closure. A closure is a function that should be evaluated in its captured environment. Representing evaluated functions as closures makes it possible to store functions on the heap.

We bind new immutable variables with the **`let`** construct. The variable we assign to can be referenced in the body of the construct, which is the term after **`in`**. A variable cannot be written to again. The assignment construct is only used for assigning values to locations. Naturally, **`let`** can be nested to bind more variables. Also note that `let x = t1 in t2` is just syntactic sugar for `(fun x => t2) t1`.

In the language we define algebraic data types using sums of products. With this feature we can write programs that mutate data structures such as linked lists and binary trees. The sum and product constructs are terms and values. We use **`left`** and **`right`** to construct a sum. The **`switch`** construct allows us to deconstruct a sum value and branch into each case of the sum. Together with the unit value we can construct, for example, a boolean data type that is either true or false as `left ()` and `right ()`. We can also construct data types with more options by right nesting sums. For example, the second option out of data type with three options would be `right (left ())`. We use `(`t`, `t`)` to construct a product or tuple of two terms and use **`fst`** and **`snd`** to deconstruct a tuple into the first and second term respectively. Again, we can right nest tuples to construct products of any size. Now we can construct the *Maybe* type in our language, which is either *none* or *some* value. We can define *none* as `left ()` and *some* as `fun x => right x`. By defining *some* as a function we have created a generic *Maybe* data type for any value of `x`.

The interesting parts of an imperative language are the constructs that change state. To best explain what they represent, let us take an example that changes state.

```
let l = ref () in
  l := !k ;
  free l ;
  free k
```

On the first line, we allocate a unit value and bind its location to `l`. Then, we create a sequence of operations with `;` by first dereferencing `k` and assigning its value to the location `l` and then deallocating the two locations. We indent the body of a **`let`** to indicate what is part of the body and what is not. Now that we have defined how to construct programs, we can define what they mean.

4.1.2 Big Step Semantics

The semantics of the imperative language are defined by the inductive relation $(t, \eta, \sigma \Downarrow v, \sigma')$, which was introduced in the previous chapter in the definition of Hoare triples. But to refresh our memory, the relation says that a term t evaluates in environment η and state σ to value v and resulting state σ'. The stateless evaluation rules are defined in Figure 4.1 as substitution-free big step semantics. This means that the evaluation rules fully evaluate a term to a value without performing explicit substitution of variables in terms. Instead, we lookup a variable in an environment (see EVAL-VAR) and extend the environment when binding a new variable (see EVAL-LET). The relation $\eta[x] = v$ states that the variable x in environment η maps to value v. We extend an environment η with a variable x that maps to a value v by $\{x \to v\} \cup \eta$, which is a new environment with the original variables from η but also where x maps to v.

We prefer big step over small step, because our Hoare triples are defined over terms that evaluate to a value, so we do not care about the intermediate steps of evaluation. Big step is also easier for implementing substitution-free evaluation, since we do not have to state the updated variable environment after the evaluation.

The language is call-by-value, which means that every term is fully evaluated before it is applied to a function. The language is also evaluated from left to right. This is important information when using state, since we pass the state from the left term to the right term. For example, we can use locations in the right term of a product that were allocated in the left term but not vice versa.

So far, we have only defined the semantics of terms that do not mutate the state. In order to define the semantics that mutate state, we could define functions that mutate the state and use them in the inductive relation. For example, to define the semantics of **`free`**, we could define a function that deallocates a location from a heap, which would look something like the following.

EVAL-FREE

$$\frac{t, \eta, \sigma \Downarrow l, \sigma_1}{\texttt{free}\ t, \eta, \sigma \Downarrow \texttt{()}, \text{deallocate}(l, \sigma_1)}$$

We first evaluate the term to a location and then update the state to remove the location. However, if we wanted to derive a Hoare triple from this rule we would have to extrinsically show some relation to Separation Logic and this function. Instead, we will use Separation Logic directly in our evaluation rules. We use the points-to relation to assert that the state of the result of evaluating the location should be separated by the location it wants to deallocate and some other state. Since we use the separating conjunction, the state separated from the points-to will no longer contain the location to be freed. See Figure 4.2 for the evaluation rules of allocation, deallocation, assignment and dereferencing.

The notation $\sigma \in H$ means that the store σ can be described by the specification H. The evaluation rules EVAL-REF and EVAL-FREE use the singleton predicate $\{\sigma\}$, which is defined

$$\textsc{eval-var}\quad \frac{\eta[x] = v}{x, \eta, \sigma \Downarrow v, \sigma} \qquad\qquad \textsc{eval-val}\quad \frac{}{v, \eta, \sigma \Downarrow v, \sigma}$$

$$\textsc{eval-product}\quad \frac{t_1, \eta, \sigma \Downarrow v_1, \sigma_1 \qquad t_2, \eta, \sigma_1 \Downarrow v_2, \sigma_2}{(\ t_1\ \texttt{,}\ t_2\ ), \eta, \sigma \Downarrow (\ v_1\ \texttt{,}\ v_2\ ), \sigma_2}$$

$$\textsc{eval-fst}\quad \frac{t, \eta, \sigma \Downarrow (\ v_1\ \texttt{,}\ v_2\ ), \sigma_1}{\texttt{fst}\ t, \eta, \sigma \Downarrow v_1, \sigma_1} \qquad\qquad \textsc{eval-snd}\quad \frac{t, \eta, \sigma \Downarrow (\ v_1\ \texttt{,}\ v_2\ ), \sigma_1}{\texttt{snd}\ t, \eta, \sigma \Downarrow v_2, \sigma_1}$$

$$\textsc{eval-left}\quad \frac{t, \eta, \sigma \Downarrow v, \sigma_1}{\texttt{left}\ t, \eta, \sigma \Downarrow \texttt{left}\ v, \sigma_1} \qquad\qquad \textsc{eval-right}\quad \frac{t, \eta, \sigma \Downarrow v, \sigma_1}{\texttt{right}\ t, \eta, \sigma \Downarrow \texttt{right}\ v, \sigma_1}$$

$$\textsc{eval-switch-left}\quad \frac{t, \eta, \sigma \Downarrow \texttt{left}\ v, \sigma_1 \qquad t_1, \{x \to v\} \cup \eta, \sigma_1 \Downarrow v_1, \sigma_2}{(\texttt{switch}\ t\ \texttt{|left}\ x\ \texttt{=>}\ t_1\ \texttt{|right}\ x\ \texttt{=>}\ t_2), \eta, \sigma \Downarrow v_1, \sigma_2}$$

$$\textsc{eval-switch-right}\quad \frac{t, \eta, \sigma \Downarrow \texttt{right}\ v, \sigma_1 \qquad t_2, \{x \to v\} \cup \eta, \sigma_1 \Downarrow v_2, \sigma_2}{(\texttt{switch}\ t\ \texttt{|left}\ x\ \texttt{=>}\ t_1\ \texttt{|right}\ x\ \texttt{=>}\ t_2), \eta, \sigma \Downarrow v_2, \sigma_2}$$

$$\textsc{eval-seq}\quad \frac{t_1, \eta, \sigma \Downarrow v_1, \sigma_1 \qquad t_2, \eta, \sigma_1 \Downarrow v_2, \sigma_2}{(t_1\ \texttt{;}\ t_2), \eta, \sigma \Downarrow v_2, \sigma_2} \qquad \textsc{eval-let}\quad \frac{t_1, \eta, \sigma \Downarrow v_1, \sigma_1 \qquad t_2, \{x \to v_1\} \cup \eta, \sigma_1 \Downarrow v_2, \sigma_2}{(\texttt{let}\ x\ \texttt{=}\ t_1\ \texttt{in}\ t_2), \eta, \sigma \Downarrow v_2, \sigma_2}$$

$$\textsc{eval-fun}\quad \frac{}{(\texttt{fun}\ x\ \texttt{=>}\ t), \eta, \sigma \Downarrow (\texttt{clos}\ \eta\ x\ \texttt{=>}\ t), \sigma} \qquad \textsc{eval-rec}\quad \frac{}{(\texttt{rec}\ f\ x\ \texttt{=>}\ t), \eta, \sigma \Downarrow (\texttt{fix}\ \eta\ f\ x\ \texttt{=>}\ t), \sigma}$$

$$\textsc{eval-fun-app}\quad \frac{t_1, \eta, \sigma \Downarrow (\texttt{clos}\ \eta'\ x\ \texttt{=>}\ t), \sigma_1 \qquad t_2, \eta, \sigma_1 \Downarrow v_1, \sigma_2 \qquad t, \{x \to v_1\} \cup \eta', \sigma_2 \Downarrow v_2, \sigma_3}{t_1\ t_2, \eta, \sigma \Downarrow v_2, \sigma_3}$$

$$\textsc{eval-rec-app}\quad \frac{t_2, \eta, \sigma_1 \Downarrow v_1, \sigma_2 \qquad \begin{array}{c} t_1, \eta, \sigma \Downarrow (\texttt{fix}\ \eta'\ f\ x\ \texttt{=>}\ t), \sigma_1 \\ t, \{x \to v_1, f \to (\texttt{fix}\ \eta'\ f\ x\ \texttt{=>}\ t)\} \cup \eta', \sigma_2 \Downarrow v_2, \sigma_3 \end{array}}{t_1\ t_2, \eta, \sigma \Downarrow v_2, \sigma_3}$$

Figure 4.1: Substitution-free big step semantics of imperative λ-calculus that do not mutate state.

$$\textsc{eval-ref}\quad \frac{t, \eta, \sigma \Downarrow v, \sigma_1 \qquad \sigma_2 \in l \mapsto v * \{\sigma_1\}}{\texttt{ref}\ t, \eta, \sigma \Downarrow l, \sigma_2} \qquad \textsc{eval-free}\quad \frac{t, \eta, \sigma \Downarrow l, \sigma_1 \qquad \sigma_1 \in l \mapsto v * \{\sigma_2\}}{\texttt{free}\ t, \eta, \sigma \Downarrow \texttt{()}, \sigma_2}$$

$$\textsc{eval-deref}\quad \frac{t, \eta, \sigma \Downarrow l, \sigma_1 \qquad \sigma_1 \in l \mapsto v * \top}{\texttt{!}\ t, \eta, \sigma \Downarrow v, \sigma_1}$$

$$\textsc{eval-assign}\quad \frac{t_1, \eta, \sigma \Downarrow l, \sigma_1 \qquad t_2, \eta, \sigma_1 \Downarrow v_2, \sigma_2 \qquad \sigma_2 \in l \mapsto v_1 * H \qquad \sigma_3 \in l \mapsto v_2 * H}{t_1\ \texttt{:=}\ t_2, \eta, \sigma \Downarrow \texttt{()}, \sigma_3}$$

Figure 4.2: Substitution-free big step semantics of imperative λ-calculus that mutate state.

as the set of stores that are equivalent to σ. This predicate is necessary for separating the eventual evaluated state by a location in EVAL-FREE and for separating the evaluated state of EVAL-REF by a new location.

The evaluation rule EVAL-DEREF does not mutate the state and only requires that the location to be dereferenced is contained in the current store.

The evaluation rule EVAL-ASSIGN is defined over a specification that is separated by the location to be updated, where the old state is separated by the old value of the location and the new state is separated by location pointing to the new value. Assignment returns the unit value.

4.2 Language Triples

Now that we have defined a syntax and semantics for our imperative language, we can define triples that are specific to the constructs of the language. As already mentioned, we derive the triples from the semantics of our language, where every evaluation rule corresponds to a triple.

If we look at our definition of triples (see Definition 3.3.2), we see that, in order to define a triple for a term, we need evidence of a value and a state to which the term evaluates and satisfy the postcondition. So, for every term we look at the evaluation rule and choose a suitable pre- and postcondition that could be satisfied by the state and return value.

4.2.1 Writing Linear Programs

There are many possibilities for conditions of a triple. A problem in defining triples is how to define triples for language constructs that have subterms. If we define triples to have arbitrary subterms, we get proof outlines that are not linear and become hard to read. Because the outlines are written down from top to bottom, reading the proof from top to bottom becomes difficult, since the steps first fully go down one branch before the next branch can be proven. A term that fits onto one line in the program might span multiple lines in the mechanized outline. This is a downside of mechanization where every step has to be explicitly stated.

A solution to this problem is to restrict the triples to be for terms in Administrative Normal Form (ANF) (Sabry and Felleisen 1992). In ANF every subterm is a variable, except for the **`let`** constructs which binds evaluated terms to new variables. This way we only have to define sequencing for the **`let`** construct and use it to evaluate terms to values before using them in other terms. To illustrate ANF and how it influences proof outlines, we take a look at the following program that swaps the values of a product allocated at location l in the store and stores the result at location k:

```
k := ref (snd (! l) , fst (! l))
```

This program is not in ANF, since the assignment to `k`, dereferencing `l` and deconstructing the product all have terms as subterms and not variables. If we wanted to write a proof outline for this program, then it would become hard to read, since there is no nice way to structure this proof and keep the program structure. It would have a tree structure that quickly becomes difficult to read based on the complexity of subterms. So, we transform the program to ANF to create a more linear program that can be read from top to bottom. In order to transform it to ANF, we create **`let`** bindings that first evaluate the terms to values and bind them to variables before use:

```
let p = ! l in
let p1 = fst p in
let p2 = snd p in
let np = (p2 , p1) in
let kv = ref np in
  k := kv
```

Now if we wanted to write a proof outline of this program we can step through the program one operation at a time and write the state in between the let bindings.

By defining triples in this form, we also restrict the set of programs that we can write outlines for. Still, it is possible to prove specifications for programs not in ANF. Since every program has an Administrative Normal Form, we can transform a non-ANF program to ANF , write a proof outline for the program in ANF and then derive a triple for the non-ANF version. If we do not change the semantics during the transformation, we can derive a triple for the original program with the following inference rule:

Lemma 4.2.1 (Coerce triple).

$$\frac{\{\eta \vdash H\}\, t'\, \{Q\} \qquad \forall v \forall \sigma \forall \sigma'.\ (t', \eta, \sigma \Downarrow v, \sigma') \Rightarrow (t, \eta, \sigma \Downarrow v, \sigma')}{\{\eta \vdash H\}\, t\, \{Q\}}\ \textsc{coerce-triple}$$

We can derive a triple for a non-ANF program by applying coerce-triple and showing that the semantics of the ANF program imply the semantics of the target non-ANF program. To see why this rule is valid, we unfold the definition of triples. We have to show that the program t satisfies postcondition Q assuming precondition H. We know that t' correctly evaluates to a value and a store that satisfies Q. Since we have the implication of evaluation, we know that if t' evaluates to a value and store, then t evaluates to that same value and store. Therefore, t also satisfies the postcondition Q and thus forms a triple with the precondition H.

Using ANF also allows us to define triples with a minimal footprint. Triples can be specified with the minimum state necessary to correctly evaluate. Since the subterm of every construct is a variable that does not change the state, we can specify the triple with minimum required state to evaluate the construct without having to worry about subterms. We can then extend the specification of these triples by the frame rule. With this design we can reuse the same triple in every outline.

VAR-TRIPLE
$$\frac{\eta[x] = v}{\{\eta \vdash \mathbf{emp}\}\, x\, \{w.\ [\![w = v]\!]\}}$$

VAL-TRIPLE
$$\frac{}{\{\eta \vdash \mathbf{emp}\}\, v\, \{w.\ [\![w = v]\!]\}}$$

PRODUCT-TRIPLE
$$\frac{\eta[x_1] = v_1 \qquad \eta[x_2] = v_2}{\{\eta \vdash \mathbf{emp}\}\, \texttt{(}\ x_1\ \texttt{,}\ x_2\ \texttt{)}\, \{w.\ [\![w = \texttt{(}\ v_1\ \texttt{,}\ v_2\ \texttt{)}]\!]\}}$$

FST-TRIPLE
$$\frac{\eta[x] = \texttt{(}\ v_1\ \texttt{,}\ v_2\ \texttt{)}}{\{\eta \vdash \mathbf{emp}\}\, \texttt{fst}\ x\, \{w.\ [\![w = v_1]\!]\}}$$

SND-TRIPLE
$$\frac{\eta[x] = \texttt{(}\ v_1\ \texttt{,}\ v_2\ \texttt{)}}{\{\eta \vdash \mathbf{emp}\}\, \texttt{snd}\ x\, \{w.\ [\![w = v_2]\!]\}}$$

LEFT-TRIPLE
$$\frac{\eta[x] = v}{\{\eta \vdash \mathbf{emp}\}\, \texttt{left}\ x\, \{w.\ [\![w = \texttt{left}\ v]\!]\}}$$

RIGHT-TRIPLE
$$\frac{\eta[x] = v}{\{\eta \vdash \mathbf{emp}\}\, \texttt{right}\ x\, \{w.\ [\![w = \texttt{right}\ v]\!]\}}$$

SWITCH-TRIPLE
$$\frac{\begin{array}{c}\eta[x] = v \\ \forall v_1.\ (v = \texttt{left}\ v_1) \Rightarrow \{\{x_1 \to v_1\} \cup \eta \vdash H\}\, t_1\, \{Q\} \\ \forall v_2.\ (v = \texttt{right}\ v_2) \Rightarrow \{\{x_2 \to v_2\} \cup \eta \vdash H\}\, t_2\, \{Q\}\end{array}}{\{\eta \vdash H\}\, \texttt{switch}\ x\ \texttt{|left}\ x_1\ \texttt{=>}\ t_1\ \texttt{|right}\ x_2\ \texttt{=>}\ t_2\, \{Q\}}$$

SEQ-TRIPLE
$$\frac{\{\eta \vdash H\}\, t_1\, \{w.\ H_1\} \qquad \{\eta \vdash H_1\}\, t_2\, \{Q\}}{\{\eta \vdash H\}\, t_1\ \texttt{;}\ t_2\, \{Q\}}$$

LET-TRIPLE
$$\frac{\{\eta \vdash H\}\, t_1\, \{Q_1\} \qquad v \Rightarrow \{\{x \to v\} \cup \eta \vdash Q_1\ v\}\, t_2\, \{Q\}}{\{\eta \vdash H\}\, \texttt{let}\ x\ \texttt{=}\ t_1\ \texttt{in}\ t_2\, \{Q\}}$$

FUN-TRIPLE
$$\frac{}{\{\eta \vdash \mathbf{emp}\}\, \texttt{fun}\ x\ \texttt{=>}\ t\, \{w.\ [\![w = (\texttt{clos}\ \eta\ x\ \texttt{=>}\ t)]\!]\}}$$

REC-TRIPLE
$$\frac{}{\{\eta \vdash \mathbf{emp}\}\, \texttt{rec}\ f\ x\ \texttt{=>}\ t\, \{w.\ [\![w = (\texttt{fix}\ \eta\ f\ x\ \texttt{=>}\ t)]\!]\}}$$

CLOS-APP-TRIPLE
$$\frac{\eta[x_1] = (\texttt{clos}\ \eta'\ x\ \texttt{=>}\ t) \qquad \eta[x_2] = v \qquad \{\{x \to v\} \cup \eta' \vdash H\}\, t\, \{Q\}}{\{\eta \vdash H\}\, x_1\ x_2\, \{Q\}}$$

FIX-APP-TRIPLE
$$\frac{\eta[x_1] = (\texttt{fix}\ \eta'\ f\ x\ \texttt{=>}\ t) \qquad \eta[x_2] = v \qquad \{\{x \to v, f \to (\texttt{fix}\ \eta'\ f\ x\ \texttt{=>}\ t)\} \cup \eta' \vdash H\}\, t\, \{Q\}}{\{\eta \vdash H\}\, x_1\ x_2\, \{Q\}}$$

Figure 4.3: Hoare triples for imperative λ-calculus without substitution that do not mutate state.

4.2.2 First Triples

We define the triples for our language in Figure 4.3, which are derived from the evaluation rules in Figure 4.1. The triples for values all have a minimal footprint of **emp**, whereas the other rules have a precondition that depends on the subterm. We observe a similarity between the semantics and the triples. The state remains the same in the pre- and postcondition, because these are the triples that do not change state, similar to the semantics. The triples that return values mention their respective return value in an equality on the return value of the term, which are equal to the values in the semantics.

The one construct that stands out is `let`. The triple LET-TRIPLE is very similar to SEQ-TRIPLE, the only difference is that the return value of the first term can be referenced in the second term. This sequencing is implemented by passing the postcondition of the first term to the precondition of the second term. Since a postcondition is also defined over a value, we quantify over the values that satisfy the postcondition to obtain a valid precondition. We can specify what the bound value should be by using an equality in a pure predicate in the postcondition of the first term. Then we can use PURE (see Lemma 3.2.6) to extract the equality and use it in our proof to reason about the bound value. Take this simple example ANF program:

```
let x = () in
  right x
```

When writing a proof outline for this program we must first apply LET-TRIPLE where the postcondition of the first term is $w.\ [\![w = \texttt{()}]\!]$. Then, when this postcondition is used in the triple of the second term, w is substituted for v, which is the value that x maps to in the environment. Then by applying PURE we can extract the equality of $v = \texttt{()}$, which implies that x maps to `()`, which we need in order to prove that the let body returns `right ()`.

4.2.3 Mutating State

The triples that remain are the triples for the constructs that operate on state. We define the triples in Figure 4.4. Again, the triples are defined for terms in ANF and have a minimal footprint.

The DEREF-TRIPLE is straightforward, since it only reads from a location and does not mutate the state specified by a points-to relation in the pre- and postcondition. If we unfold the definition of the triple, we get the formula we have to prove:

$$(l \mapsto v * H')\ \sigma \Rightarrow \exists v'.\ \exists \sigma'.\ (\texttt{!}\ \texttt{x}, \eta, \sigma \Downarrow v', \sigma') \wedge (((\llbracket v = v' \rrbracket * l \mapsto v) * H')\ \sigma')$$

If we eliminate the precondition we arrive at a goal of:

$$\exists v'.\ \exists \sigma'.\ (\texttt{!}\ \texttt{x}, \eta, \sigma \Downarrow v', \sigma') \wedge (((\llbracket v = v \rrbracket * l \mapsto v) * H')\ \sigma')$$

If we choose the value of v' to be equal to v and σ' to be equal to σ, we are left with:

$$(\texttt{!}\ \texttt{x}, \eta, \sigma \Downarrow v, \sigma) \wedge (((\llbracket v = v \rrbracket * l \mapsto v) * H')\ \sigma)$$

The left hand side of the conjunction can be constructed by applying the EVAL-DEREF rule together with EVAL-VAR and the evidence that x is contained in the environment and that σ satisfies $l \mapsto v * \top$. Since we have assumed that σ satisfies $l \mapsto v * H'$, for some frame H', σ also satisfies $l \mapsto v * \top$. Finally, the postcondition is satisfied by σ, because we can insert a pure fact of $v = v$ into our precondition.

The rule FREE-TRIPLE is another that is trivial. It follows the same line of reasoning as DEREF-TRIPLE. Our precondition is the same as DEREF-TRIPLE and we also use EVAL-VAR to evaluate

$$\textsc{ref-triple}\quad \frac{\eta[x] = v}{\{\eta \vdash \mathbf{emp}\}\ \texttt{ref}\ x\ \{w.\ \exists l.\ (\llbracket w = l \rrbracket * l \mapsto v)\}}$$

$$\textsc{free-triple}\quad \frac{\eta[x] = l}{\{\eta \vdash l \mapsto v\}\ \texttt{free}\ x\ \{w.\ \llbracket w = \texttt{()} \rrbracket\}}$$

$$\textsc{deref-triple}\quad \frac{\eta[x] = l}{\{\eta \vdash l \mapsto v\}\ \texttt{!}\ x\ \{w.\ \llbracket w = v \rrbracket * l \mapsto v\}}$$

$$\textsc{assign-triple}\quad \frac{\eta[x_1] = l \qquad \eta[x_2] = v'}{\{\eta \vdash l \mapsto v\}\ x_1\ \texttt{:=}\ x_2\ \{w.\ \llbracket w = \texttt{()} \rrbracket * l \mapsto v'\}}$$

Figure 4.4: Triples of imperative λ-calculus without substitution that mutate state.

the variable. The evaluation rule that we apply is EVAL-FREE, which requires a separate state without the location l. Since we have defined our SL triples over a frame, we can use the frame store for this instance. Then, the postcondition becomes a specification that is satisfied by the frame store, which is precisely our frame specification. Due to the way we have defined triples and our semantics, deriving this rule becomes trivial.

The triples that require some extra attention are REF-TRIPLE and ASSIGN-TRIPLE. The rule for allocation creates a new location and assignment updates a value at a location. We do not yet know how to perform these mutating operations generically for any type of store. We have seen how to reshape a store specification into another with entailment. However, we cannot use entailment to add or update locations, otherwise we could add locations at any point to any specification. Instead we need an operation to mutate a store that satisfies a specification to another that satisfies another specification. We define a new operator that instructs how to update a state to another.

Definition 4.2.2 (Update state)**.** A specification H_1 can be updated to specification H_2, written as $H_1 \Mapsto H_2$, if for every store σ that satisfies H_1 there exists a store σ' that satisfies H_2.

$$H_1 \Mapsto H_2 \quad \triangleq \quad \forall \sigma.\ H_1\ \sigma \Rightarrow \exists \sigma'.\ H_2\ \sigma'$$

With this definition we can concisely write down what it means for a store to insert or update a location. In order to derive the triples for our language and write outlines, we require that a store has these operations that we express as:

Lemma 4.2.3 (Store operations)**.**

$$\begin{array}{rcll} H & \Mapsto & \exists l.\ l \mapsto v * H & \quad *\text{-INSERT} \\ l \mapsto v * H & \Mapsto & l \mapsto v' * H & \quad *\text{-UPDATE} \end{array}$$

We define the operations on a frame to make them easy to use with REF-TRIPLE and ASSIGN-TRIPLE. The operation $*$-INSERT says there always exists a new location that can be separated with a store. $*$-UPDATE says that in any specification with a location the value can be updated. When the store operations are written down like this they neatly coincide with the pre- and postconditions of REF-TRIPLE and ASSIGN-TRIPLE. We use the frame of the Separation Logic triple to denote the frame that we perform the update operations in. Only the pure fact of equality has to added, which can be done in the same way as we did for DEREF-TRIPLE.

4.2.4 Triple to Syntax

You may have already noticed that the structure of the syntax of the imperative language arises from the triples. The triple for the product has two arguments, one for each variable, the triple for sequencing has two arguments, one triple for each subterm, and so on. The amount of arguments of the triples correspond to the amount of subterms in our syntax, except the arguments are either other triples or some evidence that the variable is contained in the environment. Our final step in writing mechanized proof outlines in the style of Hoare is to transform the triples to the same syntax as our language. So, the triple LEFT-TRIPLE becomes **`left`** $(\eta[x] = v)$ and ASSIGN-TRIPLE becomes $(\eta[x_1] = l)$ `:=` $(\eta[x_2] = v)$ and so on for each triple. I will not repeat the entire syntax here. The syntax remains the same, except now they represent triples instead of language constructs. With this syntax we can write programs and proof outlines with the same syntax. For programs we derive a program from the core constructs of the language and for a proof outline we derive a triple from the core triples of the language together with the structural rules of triples. Unfortunately, there is one imperfection in our syntax. Because we have defined function application by applying a variable to a variable, the triples CLOS-APP-TRIPLE and FIX-APP-TRIPLE require three arguments, one for each variable and a triple of the body applied to its argument. This can be solved by creating a new syntax for function application that accepts a third argument, namely the triple of the body.

4.3 Hoare Style Reasoning in Agda

Now we have defined all the rules and syntax for constructing a mechanized proof outline in our proof assistant Agda. There already exists a library[2] for defining ternary relations on algebraic structures and laws on ternary relations (Rouvoet 2021). This ternary library also defines the separating conjunction on top of the low level algebra. However, definitions for Hoare triples and its inference rules were not readily available in Agda. So, I have implemented a shallow embedding of Hoare triples with environment in Agda, together with a syntax for writing Hoare style proof outlines, which can be found in the `Relation.Hoare` module of the implementation (see Appendix A). The syntax is defined as mixfix operators using syntax declarations[3] in Agda. This mechanization of Hoare Logic needs to be instantiated with a language and its semantics and an instance of a store type. I have defined a deep embedding of the imperative language presented and its semantics in this chapter, which can be found in `Data.Lang.Lang` and `Data.Lang.Semantics`. In the module `Relation.Ternary.Constructs` and `Relation.Hoare.Constructs` you can find two instances of the store type: A functional map and a fresh list, which are shown to follow the definition and laws of the separation algebra from Section 2.4.2. I was able to derive the triples for every construct of the language, which are implemented in `Data.Lang.Hoare`, and write Hoare style proof outlines verified by Agda's dependent type system.

To illustrate what this would look like in Agda, we will write an outline for a function `swap`, that takes two locations, swaps the values of those locations and returns the unit value. The function can be specified by the following triple:

$$\{l \mapsto v_1 * k \mapsto v_2\}\,\texttt{swap l k}\,\{w.\ [\![w = \texttt{()}]\!] * l \mapsto v_2 * k \mapsto v_1\}$$

We can now write a function that can be described by the triple. An implementation of this function in ANF can be constructed as follows:

[2] https://github.com/ajrouvoet/ternary.agda

[3] https://agda.readthedocs.io/en/latest/language/syntax-declarations.html

```
fun l => fun k =>
  let t = !k in
  let s = !l in
    k := s ;
    l := t
```

First, the function dereferences the two locations and binds them to two variables. Then, the value that `l` maps to is assigned to `k` and vice versa. This function clearly satisfies the triple, but we are set out to verify this in a readable outline in Agda. This example nicely demonstrates mechanized outlines. In Chapter 6 we will perform a case study on a more complex example. The outline we would write by hand is listed in Listing 4.1 and the mechanized version in Agda is listed in Listing 4.2.

None of the syntax used in the outline is standard syntax in Agda, except for the `λ w → ...` syntax, which is used to define lambda abstractions. All other names and syntax are defined by the library.

First of all, let us point out the syntactical differences in our outline, because it does not seem to reflect our paper outline exactly. The delimiters for state are square braces instead of curly braces, because Agda restricts the use of curly braces in custom syntax, since they are part of Agda. Furthermore, the syntax of the language triples are enclosed by angles, because they would otherwise conflict with the syntax of the imperative language defined in Agda. Finally, the `let` construct is terminated by `end` to disambiguate in parsing what is part of the let body and what is not.

The first and last line of the outline specify the pre- and postcondition we have to prove. These are defined as identity functions and only serve to state the beginning and end of the outline. The arguments to the rules `frameby` and `restoreby` are given directly after the state instead of in comments, since they are now part of the proof instead of just annotations in the manual proof outline. Due to restrictions in Agda declared syntax, the names and arguments of the rules cannot be detached from the state.

The most obvious difference between the two outlines is that the mechanized outline is three times as long. In order for our proof assistant to verify our outline, we have to specify every single step. we explicitly have to state the consequence-frame rule and how to reshape the conditions, which we leave out in our manual outline, since the outline is easier to read that way.

Another difference is the handling of variables. In Agda I have implemented variables in the language with De Bruijn index (de Bruijn 1972), since it is easy to implement, but leads to difficult to read variable names. So, in the outline I use `there` and `here` as proof at what position the variable is contained in the environment. For example, on line 4 we give a proof of `here` to inform Agda that `k` is the first argument of the function and at line 15, where we assign to `k` again, the location has shifted to the third position, due to the two let bindings, which is shown by `there (there here)`.

```
fun l => fun k =>
  $\{l \mapsto v_1 * k \mapsto v_2\}$
  let s = !k in                      # s = $v_2$
  let t = !l in                      # t = $v_1$
    k := t ;
    $\{l \mapsto v_1 * k \mapsto v_1\}$              # by assignment and equality on t
    l := s
    $\{w.\ [\![w = ()]\!] * l \mapsto v_2 * k \mapsto v_1\}$  # by assignment and equality on s
```

Listing 4.1: Hoare style proof outline of swap function.

```
[ l ↦ v₁ ∗ k ↦ v₂ ]begin
⟨let s =⟩
  [ k ↦ v₂ ]frameby⟨ ∗-swap ⟩
  ⟨!⟩ here
  [ (λ w → ⟦ w ≡ v₂ ⟧ ∗ l ↦ v₁ ∗ k ↦ v₂) ]restoreby⟨ ∗-monoₗ ∗-swap ∘ ∗-assocᵣ ⟩
⟨in⟩
  [ l ↦ v₁ ∗ k ↦ v₂ ]pure⟨ s≡v₂ ⟩
  ⟨let t =⟩
    [ l ↦ v₁ ]frameby⟨ id ⟩
    ⟨!⟩ there (there here)
    [ (λ w → ⟦ w ≡ v₁ ⟧ ∗ l ↦ v₁ ∗ k ↦ v₂) ]restoreby⟨ ∗-assocᵣ ⟩
  ⟨in⟩
    [ l ↦ v₁ ∗ k ↦ v₂ ]pure⟨ t≡v₁ ⟩
    [ k ↦ v₂ ]frameby⟨ ∗-swap ⟩
    (there (there here) ⟨:=⟩ here≡ t≡v₁)
    [ (λ _ → k ↦ v₁) ]by⟨ ∗-pure⁻ˡ ⟩
    [ (λ _ → l ↦ v₁ ∗ k ↦ v₁) ]restoreby⟨ ∗-swap ⟩
    ⟨;⟩
    [ l ↦ v₁ ]frameby⟨ id ⟩
    there (there (there here)) ⟨:=⟩ there (here≡ s≡v₂)
    [ (λ w → ⟦ w ≡ unit ⟧ ∗ l ↦ v₂ ∗ k ↦ v₁) ]restoreby⟨ ∗-assocᵣ ⟩
  ⟨end⟩
⟨end⟩
[ (λ w → ⟦ w ≡ unit ⟧ ∗ l ↦ v₂ ∗ k ↦ v₁) ]∎
```

Listing 4.2: Mechanized Hoare style proof outline of swap in Agda.

Chapter 5

Proof Automation

So far, the mechanized proof outlines can be used to prove properties of programs. However, the programmer has to write every step manually in the proof, which makes the process more cumbersome than writing down an outline on paper. Furthermore, it reduces the readability of proof outlines, since it adds unnecessary clutter. In this chapter we will clean up the outlines by proof automation in Agda.

5.1 Problems from Mechanization

A problem we have with our current mechanization is that we have to manually supply a proof for every step of the reasoning when writing a proof outline. In many outlines there are steps that reshape the SL specifications to be able to use them in, for example, the frame rule. Furthermore, since we have defined the triples for our languages for terms in Administrative Normal Form, we have to supply a proof that our variable maps to a value in our environment. Recall the rules for assignment and combined consequence frame:

ASSIGN-TRIPLE

$$\frac{\eta[x_1] = l \qquad \eta[x_2] = v'}{\{\eta \vdash l \mapsto v\}\, x_1 \mathrel{:=} x_2 \,\{w.\ [\![w = (\,)]\!] * l \mapsto v'\}}$$

CONSEQUENCE-FRAME

$$\frac{H \vdash H_1 * H_2 \qquad \{\eta \vdash H_1\}\, t\, \{Q_1\} \qquad Q_1 *\cdot H_2 \mathrel{\dot{\vdash}} Q}{\{\eta \vdash H\}\, t\, \{Q\}}$$

In order to show a triple for assignment, we have to supply a proof that the variable x_1 maps to a location and the variable x_2 maps to a value. For the combined consequence frame rule, we have to give a proof of entailment for the pre- and postcondition. Both of these proof obligations add clutter to an outline that we do not write down when writing outlines on paper.

In order to illustrate how this makes outlines less readable, we take a look at an outline of the following triple:

$$\{\{x \to k, y \to 2\} \vdash l \mapsto 0 * k \mapsto 0\}\, x \mathrel{:=} y \,\{w.\ [\![w = (\,)]\!] * l \mapsto 0 * k \mapsto 2\}$$

The triple specifies that assignment of the variable y to variable x updates the value at location k and returns the unit value. The specification also includes a location l. In order to write an outline for this triple, we have to apply the consequence-frame rule first, since we have defined the triple for assignment with a minimal footprint where the precondition only contains the location to be updated. The outline in our mechanization would be written as:

```
1  ${l \mapsto 0 * k \mapsto 0}$
2  ${k \mapsto 0}$                         # frame by commutativity of *
3  x := y
4  ${w.\ [\![w = ()]\!] * k \mapsto 2}$           # assignment with x and y in environment
5  ${w.\ [\![w = ()]\!] * l \mapsto 0 * k \mapsto 2}$  # restore frame by commutativity of *
6  ${w.\ [\![w = ()]\!] * l \mapsto 0 * k \mapsto 2}$
```

Because our mechanization is precise we have to state every rule that we use and their proof obligations. We start by applying the consequence-frame rule on line 2 and supply an entailment using commutativity of $*$, because the frame must be at the right hand side of the separation. Now we have reshaped the precondition into the correct shape to apply our assignment axiom and update the value of k. We leave the proof that x and y are contained in our environment abstract, in Agda we would have to give a term that shows this fact. The value that k points to is updated to reflect the assignment. Finally, we can restore our frame again using commutativity of $*$. The highlighted lines show the difference between our mechanization and outlines on paper. So, the outline on paper would be half the length of the outline in our mechanization. We would rather remove these lines to make the proof shorter and easier to read.

The difference between our mechanization and outlines on paper can be summarized by the following two points:

1. We have to give a proof for every entailment that we use to reshape our pre- and post-conditions.

2. We have to give a proof that a variable maps to a value in our environment.

These differences cause our proof outlines to be longer and less readable. However, these proofs are necessary to make our outline complete and verifiable by Agda. The differences can be greatly reduced by use of proof automation. We take advantage of the fact that we are working in a proof assistant and instruct Agda to find a correct proof for us. Proof automation will allow us to write concise verified outlines of programs.

In the remainder of this chapter we will define proof automation to reduce the two differences that make our mechanization less readable. We will solve the first by introducing proof automation for entailment in Section 5.2 and then extending it in Section 5.3 to work for the combined consequence-frame rule. The second difference will be relieved in Section 5.4 by proof automation that finds a variable in an environment.

5.2 Automating Entailment

In this section we will design a decision procedure to decide if an entailment is valid and, when it is valid, also to automatically construct a proof. However, entailment is not decidable when allowing for all possible predicates in SL (Calcagno, Yang, and O'hearn 2001). The mechanization of SL in Agda was presented in Chapter 4 by a shallow embedding and, thus, allows for any possible predicate in set theory. This means we will have to restrict the use of proof automation in outlines to a subset of SL where entailment can be decided. We can then use this decision procedure to design a solver for a subset of SL. So, we will not be able to solve every entailment. We will see that this subset is large enough to still be very useful in proof outlines.

The subset of expressions we will be using is the separating conjunction ($*$) and the empty predicate (**emp**). We will treat all other predicates as atomic variables, which we cannot reshape. This reduces the decision problem to solving equivalences in a commutative monoid by Lemma 2.2.1. This process has been described extensively by Boutin (1997) and later by

$$\begin{tikzcd}
\llbracket P \rrbracket * (l \mapsto v * (k \mapsto w * \mathbf{emp})) \arrow[r, equal] \arrow[d, dash, "\dashv\vdash" description] & \llbracket P \rrbracket * (l \mapsto v * (k \mapsto w * \mathbf{emp})) \arrow[d, dash, "\dashv\vdash" description] \\
l \mapsto v * (k \mapsto w * \llbracket P \rrbracket) * \mathbf{emp} & (k \mapsto w * l \mapsto v) * \llbracket P \rrbracket
\end{tikzcd}$$

Figure 5.1: This diagram illustrates how we will solve an equivalence between two predicates. The two predicates in the bottom are equivalent because they have the same normal form at the top and they are equivalent to their normal form.

Grégoire and Mahboubi (2005) for the proof assistant Coq. Bove, Dybjer, and Norell (2009) have shown how to implement a solver for commutative monoids in Agda. Currently, Agda has an implementation of a noncommutative monoid solver[1] and a ring solver[2] in the standard library. We will use these ideas to implement a solver for SL in Agda.

We will not directly define a decision procedure on restricted expressions. Instead, we will create a decision procedure via a *normal form* of expressions. A normal form is a form of an expression such that any two expressions are equivalent when they have the same normal form. It is easier to decide whether two normal forms are equivalent. If we pick a sound normal form transform, such that every expression is equivalent to their normal form, we can decide whether two expressions are equivalent via their normal forms. See Figure 5.1 for an example of an equivalence that we can solve with this procedure.

The consequence of using normal forms is that we will actually create a decision procedure for equivalences of predicates in the form $H_1 \dashv\vdash H_2$. So, we will show a stronger relation than is actually necessary for the consequence rule, which requires only one part of the equivalence: $H_1 \vdash H_2$.

5.2.1 Restricted Expressions

The subset of expressions that we use can be represented by the following syntax:

$$\mathit{Pred}^{\downarrow} \quad ::= \quad \mathit{Pred}^{\downarrow} *' \mathit{Pred}^{\downarrow} \mid \mathbf{emp}' \mid x$$

Predicates will be restricted to expressions that consist of $*$, **emp** and atomic variables. Any predicate that is not one of $*$ or **emp** will be represented as atomic variables. The type of variables x will be the set of natural numbers $\mathbb{N}$ and the predicates they represent will be stored in an environment of type *Env*, which is a map of $\mathbb{N}$ to *Pred*. This will make it more convenient to compare two expressions that have the same environment and create an order on variables. This allows us to still solve entailment for expressions with arbitrary predicates, but we will handle them as atomic expressions that cannot be reshaped.

In order to convert between an expression and their restricted form, we define a macro *parse* to parse an expression to its restricted form and an environment that maps the atomic variables back to their original predicates. This macro serves as an oracle that can quote a predicate into its restricted expression. Later in Section 5.5 we will see how to define such a macro in Agda, but for now we will define it as an oracle. Let us look at an example of parsing an expression to better illustrate how this macro works:

$$\text{parse}(\llbracket P \rrbracket * l \mapsto v * \mathbf{emp}) \quad = \quad 0 *' (1 *' \mathbf{emp}') \;,\; \{0 \to \llbracket P \rrbracket, 1 \to (l \mapsto v)\}$$

[1]`https://agda.github.io/agda-stdlib/Tactic.MonoidSolver.html`

[2]`https://agda.github.io/agda-stdlib/Tactic.RingSolver.html`

The parsed expression hides the points-to and pure predicate and treats them as unknown predicates that we cannot unfold. The separation and empty predicate are parsed to their respective quoted forms. The environment maps the variables back to their original predicate.

We also define an evaluation function *eval* to convert an expression and an environment back to a predicate. Converting back to a predicate can be done by induction on the expression and looking up predicates in the environment. Furthermore, we have to require that evaluation is the inverse of parsing, in order to ensure that we can use the restricted expressions to decide entailment. If we do not have this requirement, then we can still decide whether expressions are equivalent, but never relate them to their original predicates.

Definition 5.2.1 (Evaluation of expression)**.** The function eval evaluates an expression and environment to a predicate.

$$\begin{aligned} \text{eval}(H_1^{\downarrow} *' H_2^{\downarrow}, \rho) &= \text{eval}(H_1^{\downarrow}, \rho) * \text{eval}(H_2^{\downarrow}, \rho) \\ \text{eval}(\mathbf{emp}', \rho) &= \mathbf{emp} \\ \text{eval}(x, \rho) &= \text{lookup}(x, \rho) \end{aligned}$$

Lemma 5.2.2 (Evaluation is inverse of parsing)**.** Parsing and evaluating a predicate H is equal to H.

$$\text{eval}(\text{parse}(H)) = H$$

5.2.2 Normal Forms

As stated before, we decide whether two expressions are equivalent via a *normal form*. A normal form of an expression is a sorted list of the variables in the expression. For example, the normal form of $(1 *' 0) *' (\mathbf{emp}' *' 2)$ is $(0, 1, 2)$. We define a normal form *NF* as an algebraic data type in the same structure as a list:

$$\mathit{NF} \quad ::= \quad \mathbf{nil} \mid \mathbf{cons}\ x\ \mathit{NF}$$

We also define a function *norm* for normalizing an expression. We can normalize an expression by traversing it in a left recursive manner and collecting all the variables in a list and skipping **emp′** and then sorting the list in ascending order. In essence, we flatten the tree structure of the expression to a flat list structure. We sort the list to ensure that two normal forms with equal variables, but in a different order, also have the same normal form. We can easily decide whether two normal forms are equivalent by comparing every variable in the list from left to right. Finally, we also define an evaluation function *eval*$_n$ to evaluate a normal form and environment to a predicate.

Definition 5.2.3 (Normalization of expression)**.** The function norm normalizes a quoted expression to a sorted list of variables and can be defined as follows:

$$\text{norm}(H^{\downarrow}) = \text{sort}(\text{flatten}(H^{\downarrow}))$$

$$\begin{aligned} \text{flatten}(H_1^{\downarrow} *' H_2^{\downarrow}) &= \text{concat}(\text{flatten}(H_1^{\downarrow}), \text{flatten}(H_2^{\downarrow})) \\ \text{flatten}(\mathbf{emp}') &= \mathbf{nil} \\ \text{flatten}(x) &= \mathbf{cons}\ x\ \mathbf{nil} \end{aligned}$$

$$\begin{aligned} \text{concat}(\mathbf{nil}, n_2) &= n_2 \\ \text{concat}(\mathbf{cons}\ x\ n_1, n_2) &= \mathbf{cons}\ x\ \text{concat}(n_1, n_2) \end{aligned}$$

Definition 5.2.4 (Evaluation of normal form)**.** The function eval_n evaluates a normal form and environment to a predicate.

$$\begin{aligned}\text{eval}_n(\mathbf{nil}, \rho) &= \mathbf{emp}\\ \text{eval}_n(\mathbf{cons}\ x\ n, \rho) &= \text{lookup}(x, \rho) * \text{eval}_n(n, \rho)\end{aligned}$$

5.2.3 Soundness of Normalization

In order to use the normalization in the decision procedure, the transformation must be *sound*. All evaluated normal forms of an expression should be equivalent to the evaluation of the expression. In other words, the normalization of a predicate should not change the meaning of the predicate. We can show that the normalization is sound if we also show that sorting a normal form and flattening an expression are sound. Proving that flattening is sound can be done by induction on the first argument. A sorted list can be seen as a permutation of a list and, thus, also a permutation of our expression. So, with commutativity, associativity and identity of $*$ we can show that sorting a normal form is sound. See Appendix B for the full proofs of these lemmas.

Lemma 5.2.5 (Flattening of expression to normal form is sound)**.** Evaluating a flattening of an expression $H^{\downarrow}$ to a normal form is equivalent to evaluating the expression.

$$\text{eval}_n(\text{flatten}(H^{\downarrow}), \rho) \dashv\vdash \text{eval}(H^{\downarrow}, \rho)$$

Lemma 5.2.6 (Sorting of normal form is sound)**.** Evaluating a sorted normal form n is equivalent to evaluating the normal form.

$$\text{eval}_n(\text{sort}(n), \rho) \dashv\vdash \text{eval}_n(n, \rho)$$

Lemma 5.2.7 (Normal form soundness)**.** All evaluated restricted expressions $H^{\downarrow}$ in environment ρ are equivalent to their evaluated normal form in environment ρ.

$$\text{eval}_n(\text{norm}(H^{\downarrow}), \rho) \dashv\vdash \text{eval}(H^{\downarrow}, \rho)$$

Proof. In order to show the equivalence we will start with the normalization and reason from there to the expression by equivalence of $\dashv\vdash$ (see Lemma 2.2.2). First, we unfold norm to get:

$$\text{eval}_n(\text{sort}(\text{flatten}(H^{\downarrow})), \rho)$$

Now, using the fact that sorting a normal form is sound by Lemma 5.2.6, we can transform this to:

$$\text{eval}_n(\text{flatten}(H^{\downarrow}), \rho)$$

Next we apply the soundness of flattening an expression by Lemma 5.2.5 to get:

$$\text{eval}(H^{\downarrow}, \rho)$$

which is the goal we needed to show to prove the equivalence. Therefore, evaluating a normalized expression is equivalent to evaluating the expression. □

5.2.4 Alignment of Environments

Two expressions may not have the same environment, so we cannot relate the evaluated predicates to each other. A normal form will evaluate to a different predicate under different environments. In order to decide equivalence of predicates we have to use the same environment for both sides. We can do so by aligning the right hand side of the equivalence to the environment of the left hand side of the equivalence. We define another macro *align* to rename variables from an expression in one environment to another environment. For every variable in the expression we take the predicate that it maps to in its environment and find its index in the other environment by syntactically comparing the predicates and renaming the variable to map to that predicate. See this example of aligning an expression to a different environment:

$$\text{align}(0 *' 1, \{0 \rightarrow (l \mapsto v), 1 \rightarrow [\![v \neq 0]\!]\}, \{0 \rightarrow [\![v \neq 0]\!], 1 \rightarrow (l \mapsto v)\}) = 1 *' 0$$

We assume that every predicate in an environment is syntactically unique, otherwise we might remap a variable to an incorrect predicate. This should not be an issue, since equal predicates separated by $*$ are usually false or completely useless.

We must also have that every evaluated expression in ρ_1 evaluates to the same predicate in ρ_2 when it has been aligned to ρ_2.

Lemma 5.2.8. Evaluating an expression $H^{\downarrow}$ in ρ_1 is equal to aligning to ρ_2 and evaluating the expression in ρ_2.

$$\text{eval}(H^{\downarrow}, \rho_1) = \text{eval}(\text{align}(H^{\downarrow}, \rho_1, \rho_2), \rho_2)$$

5.2.5 Decision Procedure

We have all ingredients to design a weak decision procedure for entailment of predicates. At the start of this section I showed that we were going to solve entailment via normal forms of predicate expressions. Now that we have defined a method of obtaining a normalization of an expression we can flesh out the design of Figure 5.1 with these steps into the diagram of Figure 5.2. The center design of proving equivalence via normal forms is expanded with how to obtain these normal forms.

The solving procedure is as follows. For an equivalence of two predicates, we parse both predicates to their expressions and normalize them. Then we try to align one expression to the environment of the other, to make sure they have the same atomic predicates. If this fails, then we cannot solve the entailment. But, if the normal forms are indeed equal, then we can prove the equivalence by the soundness of their normal forms. The entire proof then looks like this:

$$H_1 \dashv\vdash \text{eval}_n(\text{norm}(H_1^{\downarrow}), \rho_1) = \text{eval}_n(\text{norm}(\text{align}(H_2^{\downarrow}, \rho_2, \rho_1)), \rho_1) \dashv\vdash H_2$$

Finally, we can incorporate our decision procedure into our proof outlines. Since we can solve entailment of type $H_1 \vdash H_2$, we can use the solver to construct proofs for entailment of the pre- and postcondition in the consequence rule instead of having to supply the proof of entailment ourselves. Although the solver is restricted in what it can solve, it is still useful in constructing proofs that would be tedious to solve by hand. Take the following outline for example:

```
1  {$l \mapsto 0 * m \mapsto 7 * ([\![P]\!] * k \mapsto 2)$}
2  {$k \mapsto 2 * ([\![P]\!] * l \mapsto 0 * m \mapsto 7)$}   # consequence with solve
3  {$k \mapsto 2$}                                  # frame
4  free k
5  {emp}                                     # free axiom
6  {$[\![P]\!] * l \mapsto 0 * m \mapsto 7$}           # restore frame
```

In the consequence rule of line 2 we can now just apply solve and let the solver find a proof for us, because we know that they both separate the same predicates. So, the solver finds a proof for the entailment:

$$l \mapsto 0 * m \mapsto 7 * ([\![P]\!] * k \mapsto 2) \vdash k \mapsto 2 * ([\![P]\!] * l \mapsto 0 * m \mapsto 7)$$

$$\begin{tikzcd}
& H_1' \arrow[r, equal, "\text{decidable normal form equivalence}"] \arrow[dd, equal, "\text{soundness}"] & H_2' \arrow[dd, equal, "\text{soundness}"] & H^{\downarrow}_{2\to 1}, \rho_1 \arrow[l, "\text{norm}"'] \\
H^{\downarrow}_1, \rho_1 \arrow[ur, "\text{norm}"] & & & \\
& H_1 \arrow[ul, "\text{parse}"] & H_2 \arrow[r, "\text{parse}"] & H^{\downarrow}_2, \rho_2 \arrow[uu, "\text{align to } \rho_1"]
\end{tikzcd}$$

Figure 5.2: This diagram illustrates how to solve an equivalence between two predicates depicted at the bottom. You can read the diagram by starting at H_1 in the bottom left, then, by the soundness of normal forms, move up to arrive at the normal form of H_1. From here, you go right to the equal normal form of H_2 and use the soundness of normal forms in reverse to arrive at the equivalent predicate H_2. The other paths show how to obtain the equivalent normal forms.

This is a huge improvement over having to supply this proof ourselves. The proof for this entailment would be significantly larger than using the solver and we would also have to give a proof of restoring the frame, resulting in two large tedious proofs.

Unfortunately, we still have to separate the consequence and the frame, because our solver is not capable of solving the entailment necessary for the combined consequence frame rule: $H \vdash H_1 * H_2$ for an unknown H_2. The solver would not be able to find a proof, because the left hand side of the entailment does not contain the predicate H_2. We could just supply the necessary frame and invoke the solver, but we can do better by extending our solver to find this frame for us.

5.3 Automating Framing

In this section we will extend our solver to first find the unknown frame and then do the usual process of solving we have defined previously. So, reusing the previous example, our goal is to write down a proof outline like this:

```
1  {$l \mapsto 0 * m \mapsto 7 * (⟦P⟧ * k \mapsto 2)$}
2  {$k \mapsto 2$}                      # frame with solve
3  free k
4  {emp}                                # free axiom
5  {$⟦P⟧ * l \mapsto 0 * m \mapsto 7$}     # restore frame with solve
```

On line 2 we use the consequence frame rule without supplying a frame and use the solver to find a proof for the entailment. This would require our solver to find an entailment of:

$$l \mapsto 0 * m \mapsto 7 * (\llbracket P \rrbracket * k \mapsto 2) \vdash k \mapsto 2 * H$$

where H is an unknown frame that the extended frame solver will figure out automatically, before it can do the actual solving. In restoring the frame on line 5 we no longer need to find the frame, since it is the same as the one we found on line 2. So, we can restore the frame using the regular solver.

We will derive the unknown frame from the known predicates. The fragment $k \mapsto 2$ we want to use is contained in the separation we started with. So, in order to get the frame, we remove $k \mapsto 2$ from the separation and set this to the frame H to get the entailment:

$$l \mapsto 0 * m \mapsto 7 * (\llbracket P \rrbracket * k \mapsto 2) \vdash k \mapsto 2 * (l \mapsto 0 * m \mapsto 7 * \llbracket P \rrbracket)$$

which we can solve with our solver. So, in general for an entailment $H \vdash H_1 * H_2$, we substitute H_2 with H_1 subtracted from H and then solve the resulting entailment. We have already defined a solver that was able to find a proof for entailment given that the normal forms were equal. For solving with an unknown frame, we will define a solver that finds a proof given that the fragment H_1 is contained in H.

5.3.1 Finding a Frame

The first step is finding the right frame. We will define the frame as the difference between the specification we start with and the smaller specification we want to use. We define this operation on the normal forms of expressions, since these are sorted lists, where the expressions are represented by easy to compare natural numbers. We define the function *subtract*, which subtracts a normal form from another normal form. The following example demonstrates how the function operates:

$$\text{subtract}(\textbf{cons}\ 0\ (\textbf{cons}\ 1\ (\textbf{cons}\ 2\ \textbf{nil})), \textbf{cons}\ 1\ \textbf{nil}) = \textbf{cons}\ 0\ (\textbf{cons}\ 2\ \textbf{nil})$$

The second normal form is subtracted from the first and we are left with the first normal form where the variable 1 has been removed.

But what if the second normal contained a variable that is not contained in the first? The function would fail, since the solver cannot solve an entailment where a variable is in the right hand side but not in the left hand side. Therefore, the entire solving process will fail. This function only succeeds if the second normal form is contained in the first. So, we define a decidable relation on normal forms that defines when a normal form is contained in another.

Definition 5.3.1 (Normal form subset). An normal form n_1 is a subset of normal form n_2, written as $n_1 \subseteq n_2$, when all variables of n_1 are contained in n_2 in order.

Lemma 5.3.2 (Subset decidable). $n_1 \subseteq n_2$ is decidable, for any two normal forms n_1 and n_2.

Definition 5.3.3 (Subtract normal forms). Given two normal forms n_1 and n_2, where $n_2 \subseteq n_1$, then subtract(n_1, n_2) subtracts all variables of n_2 from n_1.

$$\begin{aligned}
\text{subtract}(n_1, \textbf{nil}) &= n_1 \\
\text{subtract}(\textbf{cons}\ x_1\ n_1, \textbf{cons}\ x_2\ n_2) &= \textbf{cons}\ x_1\ (\text{subtract}\ n_1\ (\textbf{cons}\ x_2\ n_2)) && \text{if } x_1 \neq x_2 \\
\text{subtract}(\textbf{cons}\ x_1\ n_1, \textbf{cons}\ x_2\ n_2) &= \text{subtract}(n_1, n_2) && \text{if } x_1 = x_2
\end{aligned}$$

We define the subtract function only on two normal forms, where the second is a subset of the first. If n_2 is empty, subtract is defined as n_1. If n_2 contains a variable, then n_1 must also contain a variable, otherwise n_2 would not be a subset of n_1. If the variables are not equal then the variable from n_2, must be contained somewhere later in n_1, so, we leave in the variable from n_1 and subtract n_2 from the tail of n_1. If the variables are equal then we remove the variable and subtract the tails.

5.3.2 Frame Soundness

Now that we have defined how to subtract two normal forms, we can define the soundness property. We are still working with normal forms, so, soundness is defined as an equivalence on evaluated normal forms:

Lemma 5.3.4 (Frame soundness)**.** If $n_2 \subseteq n_1$, then evaluating n_1 is equivalent to evaluating n_2 and separately subtract(n_1, n_2).

$$\text{eval}_n(n_1, \rho) \dashv\vdash \text{eval}_n(n_2, \rho) * \text{eval}_n(\text{subtract}(n_1, n_2), \rho)$$

Proof. We will prove soundness of frame equivalence by induction on n_2.

The base case is when n_2 equals **nil**. The equivalence becomes:

$$\text{eval}_n(n_1, \rho) \dashv\vdash \text{eval}_n(\mathbf{nil}, \rho) * \text{eval}_n(\text{subtract}(n_1, \mathbf{nil}), \rho)$$

If we simplify from the definitions of *eval* and *subtract* we get a right hand side of:

$$\mathbf{emp} * \text{eval}_n(n_1, \rho)$$

Then by left identity of $*$ the right hand side becomes:

$$\text{eval}_n(n_1, \rho)$$

which is equivalent to the left hand side of the equivalence. Therefore, if n_2 equals **nil** then *subtract* is sound.

The first inductive case is when n_2 equals **cons** $x_2\ n_2$ and n_1 equals **cons** $x_1\ n_1$, where $x_1 \neq x_2$. The equivalence then becomes:

$$\text{eval}_n(\mathbf{cons}\ x_1\ n_1, \rho) \dashv\vdash \text{eval}_n(\mathbf{cons}\ x_2\ n_2, \rho) * \text{eval}_n(\text{subtract}(\mathbf{cons}\ x_1\ n_1, \mathbf{cons}\ x_2\ n_2), \rho)$$

By definition of *eval$_n$* we can simplify the left hand side to:

$$\text{lookup}(x_1, \rho) * \text{eval}_n(n_1, \rho)$$

Now we can apply the induction hypothesis on eval$_n(n_1, \rho)$ by monotonicity of $*$ to arrive at:

$$\text{lookup}(x_1, \rho) * \text{eval}_n(\mathbf{cons}\ x_2\ n_2, \rho) * \text{eval}_n(\text{subtract}(n_1, \mathbf{cons}\ x_2\ n_2), \rho)$$

Then we reorder the separation to:

$$\text{eval}_n(\mathbf{cons}\ x_2\ n_2, \rho) * \text{lookup}(x_1, \rho) * \text{eval}_n(\text{subtract}(n_1, \mathbf{cons}\ x_2\ n_2), \rho)$$

which we can fold by definition of *eval$_n$* back to:

$$\text{eval}_n(\mathbf{cons}\ x_2\ n_2, \rho) * \text{eval}_n(\text{subtract}(\mathbf{cons}\ x_1\ n_1, \mathbf{cons}\ x_2\ n_2), \rho)$$

which is equal to the right hand side, thus showing the first inductive case.

The second inductive case is when $x_1 = x_2$. The equivalence becomes the same as previous case:

$$\text{eval}_n(\mathbf{cons}\ x_1\ n_1, \rho) \dashv\vdash \text{eval}_n(\mathbf{cons}\ x_2\ n_2, \rho) * \text{eval}_n(\text{subtract}(\mathbf{cons}\ x_1\ n_1, \mathbf{cons}\ x_2\ n_2), \rho)$$

Again we simplify the left hand side and apply the induction hypothesis to get:

$$\text{lookup}(x_1, \rho) * \text{eval}_n(n_2, \rho) * \text{eval}_n(\text{subtract}(n_1, n_2), \rho)$$

Then we re-associate the separation to:

$$(\text{lookup}(x_1, \rho) * \text{eval}_n(n_2, \rho)) * \text{eval}_n(\text{subtract}(n_1, n_2), \rho)$$

And since $x_1 = x_2$, we can simplify to:

$$\text{eval}_n(\mathbf{cons}\ x_2\ n_2, \rho) * \text{eval}_n(\text{subtract}(\mathbf{cons}\ x_1\ n_1, \mathbf{cons}\ x_2\ n_2), \rho)$$

Which is equal to the right hand side of the equivalence. Therefore, since we have shown the equivalence for the base case and two inductive cases, subtraction of normal forms is sound. □

5.3.3 Decision Procedure

Armed with frame soundness we can design a weak decision procedure for solving entailment with unknown frame. For an equivalence of two predicates and an unknown frame, we parse both predicates to their expressions and normalize them. We cannot parse the frame, since it is not yet known what it should be. Then, we try to align one expression to the environment of the other, to make sure they evaluate in the same environment. If this fails, then we cannot solve the entailment. Next, we check if the normal form of the predicate on the right hand side is a subset of the left hand side. If not we fail the procedure and cannot find a solution. But, if it is a subset, then we set the unknown frame to the difference of the left hand and right hand side by *subtract* and use the soundness property to prove the equivalence.

We can use this decision procedure to automatically solve entailments for the consequence-frame rule without giving an explicit frame. Now we can write outlines like the one from the beginning of the section:

```
{$l \mapsto 0 * m \mapsto 7 * (\llbracket P \rrbracket * k \mapsto 2)$}
{$k \mapsto 2$}                          # frame with solve
free k
{$\mathbf{emp}$}                         # free axiom
{$\llbracket P \rrbracket * l \mapsto 0 * m \mapsto 7$}       # restore frame with solve
```

Since we know that $k \mapsto 2$ is contained in $l \mapsto 0 * m \mapsto 7 * (\llbracket P \rrbracket * k \mapsto 2)$, we can use the frame solver to find an entailment for an unknown frame. This reduces the amount of proofs that are necessary for writing the proof outline, since it is no longer required to explicitly show how to obtain the fragment from the complete specification. Furthermore, the outline becomes more readable, since tedious details are left out.

5.4 Automating Variable Lookup

Another problem we set out to solve was the inconvenience of manually providing evidence that a variable is contained in an environment. Solving this problem is specific to the representation of variables in a language. As was already mentioned in Section 4.3, the mechanization in Agda uses De Bruijn index for representing variables and a list for the environment. So, variables are represented by their index in the environment. For example, an index of zero indicates the value of the last variable added to the environment. In Agda, a triple with variables looks like:

$$\{\mathbf{cons}\ l\ (\mathbf{cons}\ v\ (\mathbf{cons}\ k\ \mathbf{nil})) \vdash l \mapsto v\}\ \texttt{var 0 := var 2}\ \{l \mapsto k\}$$

The environment is represented as a list in the same structure as a normal form and variables are represented as `var n`, where `n` is the index in the environment. In this example triple, we assign the variable with index 2 to the location represented by index 0, which results in updating the location l to the value k. This triple can be proven using the assign-triple rule:

$$\text{ASSIGN-TRIPLE}\quad \frac{\eta[x_1] = l \qquad \eta[x_2] = v'}{\{\eta \vdash l \mapsto v\}\ x_1 \texttt{ := } x_2\ \{l \mapsto v'\}}$$

If we apply the rule, the proof obligation becomes:

$$\mathbf{cons}\ l\ (\mathbf{cons}\ v\ (\mathbf{cons}\ k\ \mathbf{nil}))[0] = l \quad \text{and} \quad \mathbf{cons}\ l\ (\mathbf{cons}\ v\ (\mathbf{cons}\ k\ \mathbf{nil}))[2] = k$$

To prove these two statements, we use two inference rules: either the value is at location zero or the value is at a larger index in the list. These rules can be formalized as follows:

Definition 5.4.1 (Rules for indexing value in environment).

$$\frac{}{(\textbf{cons}\ v\ \eta)[0] = v}\ \textsc{here} \qquad \frac{\eta[n] = v}{(\textbf{cons}\ v'\ \eta)[1+n] = v}\ \textsc{there}$$

Now we can fill in the proof obligations. The first can be solved directly by HERE and for the value k at position 2, we have to apply THERE twice and then HERE.

This process of giving these proofs can be very tedious and it is not clear to a reader what is meant by the THERE and HERE rules. We can automate these proofs by looking at the index n of the variable and applying the THERE rule n times and then apply the HERE rule.

5.5 Proof Automation in Agda

We have seen how to introduce proof automation to mechanized proof outlines in an abstract way. Now we will see how to achieve proof automation in the case of Agda. Proof automation in Agda can be implemented by reflection (van der Walt and Swierstra 2012). Reflection is the representation of Agda terms as data structures in Agda. This allows the programmer to write programs that write programs. Because programs correspond to proofs in Agda, we can use reflection to construct proofs. Reflection is achieved through *quoting* and *unquoting* of terms. Quoting is parsing a term in Agda to a data structure in Agda, and unquoting is the inverse of quoting.

A program writing a program is known as a *macro*[3] in Agda. When a macro is encountered during type-checking in a program, then Agda will call the macro with the goal type as argument and other arguments that were given to the macro. Then, the macro will construct a term to satisfy the goal. Type-checking succeeds when all goals have been solved, but can also fail when there are unsolved goals or other user specified errors.

I have implemented the functions, data structures and soundness lemmas from the previous sections in Agda to achieve proof automation. See Appendix A for a reference of the implementation in Agda. All implementation details of proof automation with respect to entailment can be found in the `Relation.Ternary.Tactic` module. The implementation details of automating variable lookup can be found in the `Data.Lang.Tactic` module. The *parse* and *align* macros have been implemented as macros in Agda, which can be found in the module `Expression`, and the final decision procedure for solving entailment and framing is also a macro, which can be found in `Core`. The lemmas for proving soundness are implemented in the `Expression.Properties` and `CommutativeMonoidSolver` modules. Finally, I declared new syntax for using the macros in proof outlines, which can be found in `Relation.Hoare`.

To see what the proof outlines look like with this new syntax, we take another look at the swap example from the previous chapter Section 4.3, but now with proof automation. The new outline is listed in Listing 5.2 and, for reference, the outline without proof automation is listed again in Listing 5.1. Now we can use `auto`, which is defined by the library, to inform Agda to find a frame and solve the entailment for us, using the combined consequence-frame rule. The only manual entailment that is still necessary is removing the pure fact on the return value of assigning to `k` on line 8. Furthermore, we use the `var!` macro, which is also defined by the library, to tell Agda to look for a variable in the environment that points to the given value. For example, on line 2 we require a variable that maps to `k`. On line 6 we see how Agda uses the equality we extracted on line 5 to find the variable that maps to the value that l points to.

[3]`https://agda.readthedocs.io/en/v2.6.2.1/language/reflection.html#macros`

```
[ l ↦ v₁ ∗ k ↦ v₂ ]begin
⟨let x =⟩
  [ k ↦ v₂ ]frameby⟨ ∗-swap ⟩
  ⟨!⟩ here
  [ (λ w → ⟦ w ≡ v₂ ⟧ ∗ l ↦ v₁ ∗ k ↦ v₂) ]restoreby⟨ ∗-monoₗ ∗-swap ∘ ∗-assocᵣ ⟩
⟨in⟩
  [ l ↦ v₁ ∗ k ↦ v₂ ]pure⟨ x≡v₂ ⟩
  ⟨let y =⟩
    [ l ↦ v₁ ]frameby⟨ id ⟩
    ⟨!⟩ there (there here)
    [ (λ w → ⟦ w ≡ v₁ ⟧ ∗ l ↦ v₁ ∗ k ↦ v₂) ]restoreby⟨ ∗-assocᵣ ⟩
  ⟨in⟩
    [ l ↦ v₁ ∗ k ↦ v₂ ]pure⟨ y≡v₁ ⟩
    [ k ↦ v₂ ]frameby⟨ ∗-swap ⟩
    (there (there here) ⟨:=⟩ here≡ y≡v₁)
    [ (λ _ → k ↦ v₁) ]by⟨ ∗-pure⁻ˡ ⟩
    [ (λ _ → l ↦ v₁ ∗ k ↦ v₁) ]restoreby⟨ ∗-swap ⟩
    ⟨;⟩
    [ l ↦ v₁ ]frameby⟨ id ⟩
    there (there (there here)) ⟨:=⟩ there (here≡ x≡v₂)
    [ (λ w → ⟦ w ≡ unit ⟧ ∗ l ↦ v₂ ∗ k ↦ v₁) ]restoreby⟨ ∗-assocᵣ ⟩
  ⟨end⟩
⟨end⟩
[ (λ w → ⟦ w ≡ unit ⟧ ∗ l ↦ v₂ ∗ k ↦ v₁) ]∎
```

Listing 5.1: Hoare style proof outline of swap in Agda without proof automation.

```
[ l ↦ v₁ ∗ k ↦ v₂ ]begin
⟨let x =⟩ auto (⟨!⟩ var! (loc k)) ⟨in⟩
  [ l ↦ v₁ ∗ k ↦ v₂ ]pure⟨ x≡v₂ ⟩
  ⟨let y =⟩ auto (⟨!⟩ var! (loc l)) ⟨in⟩
    [ l ↦ v₁ ∗ k ↦ v₂ ]pure⟨ y≡v₁ ⟩
    auto (var! (loc k) ⟨:=⟩ var! v₁)
    [ (λ w → ⟦ w ≡ unit ⟧ ∗ l ↦ v₁ ∗ k ↦ v₁) ]by⟨⟩
    [ (λ _ → l ↦ v₁ ∗ k ↦ v₁) ]by⟨ ∗-pure⁻ˡ ⟩
    ⟨;⟩
    [ l ↦ v₁ ∗ k ↦ v₁ ]by⟨⟩
    auto (var! (loc l) ⟨:=⟩ var! v₂)
  ⟨end⟩
⟨end⟩
[ (λ w → ⟦ w ≡ unit ⟧ ∗ l ↦ v₂ ∗ k ↦ v₁) ]∎
```

Listing 5.2: Hoare style proof outline of swap in Agda with proof automation.

Chapter 6

Case Study

The mechanization for writing Hoare style proof outlines in the proof assistant Agda is complete. In Section 4.3 we have already looked at a simple example in our imperative language. In this chapter we will study a more complex example of copying a list structure in memory and see if the outline is still readable.

6.1 Copy List Implementation

Before we can start constructing a program that copies a list, we need a data structure to represent a list. We will represent the list structure as an algebraic data type with two constructors: `nil` for the empty list and `cons` for an element prepended to a list. In our imperative language (see Section 4.1) we can create two new definitions:

```
nil  = left ()
cons = fun x => fun xs => right (x, xs)
```

Since our language does not support new type definitions, we bind the definitions in our metatheory. We define `nil` as the left sum of the unit value, since the empty list does not carry any value. `cons` is defined as a function with two arguments, where the first argument is the value at the head of the list and the second argument the tail of the list. In order to construct a list in the store, we pass a value as the first argument and a location pointing to another list in the store as the second argument to `cons`. This creates a single linked list structure in the store. For example, a list of three values can be specified by:

$$l_1 \mapsto (\texttt{cons}\ v_1\ l_2) * l_2 \mapsto (\texttt{cons}\ v_2\ l_3) * l_3 \mapsto (\texttt{cons}\ v_3\ l_4) * l_4 \mapsto \texttt{nil}$$

Naturally, we can define new Hoare triples for these constructors, which can be derived from existing triples of product, sum and function application (see Figure 4.3):

Lemma 6.1.1 (List constructor Hoare triples).

$$\text{NIL-TRIPLE}\quad \frac{}{\{\eta \vdash \mathbf{emp}\}\ \texttt{nil}\ \{w.\ [\![w = \texttt{left}\ ()]\!]\}} \qquad \text{CONS-TRIPLE}\quad \frac{\eta[x_1] = v_1 \qquad \eta[x_2] = v_2}{\{\eta \vdash \mathbf{emp}\}\ \texttt{cons}\ x_1\ x_2\ \{w.\ [\![w = \texttt{right}\ (\ v_1\ ,\ v_2\ )]\!]\}}$$

Now we can construct a program that takes a list data structure allocated in the store and copies it by allocating a new list data structure. We implement the program as a recursive function in Listing 6.1. We write the program in ANF to ease the process of writing outlines as discussed in Section 4.2. The function takes one value of a list, which is not the location of a list. Since the list is a sum, we can match to create two branches: copying the empty list and copying a value and a list. If the list is empty then it is not allocated and we can

```
copy-list = rec f v =>
  switch v
    | left  v1 => nil
    | right v1 =>
      let l1 = snd v1 in
      let u1 = ! t in
      let u2 = f t in
      let l2 = ref u2 in
      let v2 = fst v1 in
        cons v2 l2
```

Listing 6.1: Copy list function.

return `nil`. If the list is a value and a tail, we dereference the location of the tail and call the recursive function to copy the tail. Finally, the function allocates a new location that points to the copied list and constructs a new list of the original value and the freshly allocated copied list.

6.2 Copy List Specification

The next step in writing a proof outline for this program is to specify a triple for the application of the `copy-list` function. We need a store predicate that specifies what a list structure looks like in the store. We design a new store predicate IsList(v, x) which specifies that a list value v in our language is represented by the mathematical list x, which is a list structure from our metatheory. In order to differentiate between the list from the language and the list from the metatheory, we define two new meta list constructors: $[]$ for the empty list and $v :: x$ for constructing a list from a value v and list x. We can derive IsList from already existing store predicates. An allocated list is a separated chain of locations, where each location points to a value and the next location. For example, the specification of the list $v_1 :: v_2 :: v_3 :: []$ would look like:

$$\text{IsList}(\texttt{cons}\ v_1\ l_2, v_1 :: v_2 :: v_3 :: []) = l_2 \mapsto (\texttt{cons}\ v_2\ l_3) * l_3 \mapsto (\texttt{cons}\ v_3\ l_4) * l_4 \mapsto \texttt{nil}$$

We define the IsList predicate inductively on the list from the language. If the list is `nil` then the store is empty and the meta list is equal to $[]$. If the list is `cons` then the store contains a location that points to the next list and the meta list is equal to the current value and the tail, which is again specified by IsList.

Lemma 6.2.1 (IsList predicate)**.** The IsList(v, x) predicate inductively relates value v in the language to a meta list structure x with constructors $[]$ and $x :: y$.

$$\textsc{IsList-nil}\ \frac{\textbf{emp}}{\text{IsList}(\texttt{left ()}, [])} \qquad \textsc{IsList-cons}\ \frac{l \mapsto v_1 * \text{IsList}(v_1, y)}{\text{IsList}(\texttt{right}\ (v,\ l), v :: y)}$$

Now we can define the Hoare triple of the `copy-list` function. The precondition is a list specification of the value passed to `copy-list` represented by some list x. After execution the return value should be specified by the same list x and separately the original list. The separating conjunction ensures that the freshly allocated list is completely separate from the original list and the function does not just return the list immediately. Moreover, the copied list has the same structure, since they are both specified by the same list of values. We express the triple of `copy-list` as follows:

```
copy-list = rec f v =>
  {IsList(v, x)}
  switch v
    | left  v1 =>
      {IsList(left (), x)}
      {emp}                        # unfold IsList with x = []
      nil
      {w. ⟦w = left ()⟧}
      {w. IsList(w, [])}           # by definition of IsList
      {w. IsList(w, []) * emp}     # by right identity of *
      {w. IsList(w, x) * IsList(v, x)} # by definition of IsList and x = []
    | right v1 =>
      {IsList(right (v2, l1), x)}
      {l₁ ↦ u₁ * IsList(u₁, y)}                      # unfold list with x = v2 :: y
      let l1 = snd v1 in
      let u1 = ! l1 in
      {IsList(u₁, y)}                                 # frame
      let u2 = f t in
      {IsList(u₂, y) * IsList(u₁, y)}                 # induction hypothesis
      {IsList(u₂, y) * l₁ ↦ u₁ * IsList(u₁, y)}       # restore frame
      let l2 = ref u2 in
      {l₂ ↦ u₂ * IsList(u₂, y) * l₁ ↦ u₁ * IsList(u₁, y)}
      let v2 = fst v1 in
        cons v2 l2
      {w. ⟦w = right (v₂, l₂)⟧ * l₂ ↦ u₂ * IsList(u₂, y) * l₁ ↦ u₁ * IsList(u₁, y)}
      {w. IsList(w, x) * l₁ ↦ u₁ * IsList(u₁, y)}    # by definition of IsList
      {w. IsList(w, x) * IsList(v, x)}               # by definition of IsList
```

Listing 6.2: Hoare style proof outline of copy list function.

Lemma 6.2.2 (Copy list Hoare triple).

$$\frac{}{\{\eta \vdash \text{IsList}(v, x)\}\ \texttt{copy-list v}\ \{w.\ \text{IsList}(w, x) * \text{IsList}(v, x)\}}\ \textsc{copy-list-triple}$$

6.3 Copy List Outline

The final step is writing the actual proof outline. Listing 6.2 lists the program annotated with a proof outline and Listing 6.3 lists the mechanized proof outline in Agda. I implemented the IsList predicate as an inductive data type in Agda together with some useful lemmas that you see mentioned in the consequence rules. The full implementation details can be found in the `LangHoareOutlines.CopyList` module in the `examples` directory (see Appendix A).

The outline begins by applying the rule for **`switch`**. The mechanization in Agda first requires to deconstruct the value `v` into a sum, since it is not directly clear to Agda that the value must be a sum. Since IsList is only defined for **`left`** and **`right`** it follows that `v` is a sum. First we reason about the **`left`** branch. If the given value is **`left`** then it follows that the value must be `nil`. Again, this line of reasoning is not directly clear to Agda, so we provide a lemma that this is the case. Then, we unfold the definition of IsList into **emp**, from which follows that the meta list x must also be empty. In Agda we carry this proof using pure predicates and then match on the equality using the pure rule. In the mechanization on line 10 we match on `refl`, which means that Agda will recognize that x equals the empty list. Then we can apply the triple for `nil` (see Lemma 6.1.1), since the precondition of **emp** matches. To finish this branch of reasoning, we reshape the postcondition into the required two separate lists. Since, the lists are empty it follows from the definition of IsList-nil that the return value is `nil`, therefore, if the given list is empty then the copy is also empty. Next we show the other branch, where the given list is a `cons`. Again, we need some extra lines in the mechanization to show Agda that IsList of **`right`** implies that the list is a `cons`. This deconstruction introduces some new variables into the proof. In Agda we introduce them through the existential quantifier, whereas in the manual outline we just create them from thin air. This is not a problem, because through the exists rule we can lift them outside the specification. I implemented an n-ary exists rule in Agda to extract n existential quantifiers in one line, instead of n lines, which can be found in the `Relation.Hoare` module. Then, similar to the branch of `nil`, we unfold the definition of IsList into a location and another list. After the unfolding follows a sequence of let bindings. In the mechanization we use a combination of solver calls and the pure rule to extract the substitution of the variable. We arrive at the recursive call, where we first frame the location and apply the induction hypothesis to gain a copy of the tail of the list. This is sub-structural recursion, so the proof is terminating, which is recognized by the Agda termination checker. After the recursive call we restore the frame. Finally, we copy the list by allocating a new location for the tail and combining it with the head value in a product. By definition of IsList-cons we can fold the location and IsList predicate back into a single list. We have shown that both branches of the `switch` result in a copied list, therefore, the function `copy-list` returns a copy of the given list.

The mechanization tries to make use of the solver where possible. However, at line 49, 57, 59 and 60 we have not used the solver. This is because Agda is not able to terminate type checking in reasonable amount of time in these cases. Type checking does not terminate within 10 minutes, which is an unreasonable amount of time when interactively writing proof outlines. This was tested on an Intel Core i7-11800H with 16GB of RAM with Agda version 2.6.2. Usually you type check the proof after every step. A type checked file is also necessary in order to get assistance from Agda on proof obligations. If the proofs are provided manually without solver calls, type checking takes an average of 15 seconds, which is still long, but feasible. The reason that the type checker takes an unreasonable amount of time is because of the size of the predicates. Solving an entailment of a separation of three atomic predicates is reasonable for the solver as demonstrated by line 39. However, a separation of five or more atomic variables exponentially increases to an infeasible amount of time for writing interactive outlines. The final step of the solve macro, the unification of the produced proof and the goal, takes the most amount of time and is also the step which increases exponentially in time by the size of the input. The other steps of parsing, alignment and finding the frame are negligible on these input sizes. This is a limitation of reflection in Agda and it is not certain what causes it.

```
[ IsList v xs ]begin
[ ∃[ v′ ] (⟦ v ≡ sum v′ ⟧ ∗ IsList v xs) ]by⟨ list-is-sum ⟩
[ _ ]exists⟨ v′ ⟩
[ IsList v xs ]pure λ { refl →
[ IsList (sum v′) xs ]by⟨⟩
⟨switch⟩ var! v
  |left⇒ (λ { {v₁} v₁≡nil@refl →
    [ IsList (sum (inj₁ v₁)) xs ]by⟨⟩
    [ ⟦ v₁ ≡ unit × xs ≡ []ˡ ⟧ ]by⟨ unnil ⟩
    []pure-emp (λ { (refl , refl) →
    ⟨nil⟩
    [ (λ w → ⟦ w ≡ nil-val ⟧) ]by⟨⟩
    [ (λ w → IsList w []ˡ) ]by⟨ mknil ⟩
    [ (λ w → IsList w []ˡ ∗ Emp) ]byauto
    [ (λ w → IsList w []ˡ ∗ IsList nil-val []ˡ) ]by⟨ ∗-monoₗ emp-is-nil ⟩
    [ (λ w → IsList w xs ∗ IsList v xs) ]∎ })})
  |right⇒ (λ { {v₂} v₂≡cons@refl →
    [ IsList (sum (inj₂ v₂)) xs ]by⟨⟩
    [ ∃[ l₁ ] ∃[ x ] ∃[ t₁ ] ∃[ ys ] (⟦ v₂ ≡ prod (x , loc l₁) × xs ≡ x ::ˡ ys ⟧ ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]by⟨ uncons ⟩
    [ _ ]existsₙ⟨ 4 ⟩ (λ { (l₁ , x , t₁ , ys) →
    [ ⟦ v₂ ≡ prod (x , loc l₁) × xs ≡ x ::ˡ ys ⟧ ∗ l₁ ↦ t₁ ∗ IsList t₁ ys ]by⟨⟩
    [ l₁ ↦ t₁ ∗ IsList t₁ ys ]pure (λ { (refl , refl) →
    ⟨let _ =⟩
      [ Emp ]auto
      (⟨snd⟩ var! v₂)
      [ (λ w → ⟦ w ≡ loc l₁ ⟧ ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]autorestore
    ⟨in⟩
      [ l₁ ↦ t₁ ∗ IsList t₁ ys ]pure⟨ l₁≡ ⟩
      ⟨let _ =⟩
        [ l₁ ↦ t₁ ]auto
        (⟨!⟩ var! (loc l₁))
        [ (λ w → ⟦ w ≡ t₁ ⟧ ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]autorestore
      ⟨in⟩
        [ l₁ ↦ t₁ ∗ IsList t₁ ys ]pure⟨ t₁≡ ⟩
        ⟨let t₂ =⟩
          [ IsList t₁ ys ]auto
          var! (fix η copy-body) ⟨•⟩ var! t₁
          ⟨rec-body⟩ copy-body-triple t₁ ys
          [ (λ w → IsList w ys ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]autorestore
        ⟨in⟩
          ⟨let a =⟩
            [ Emp ]auto
            (⟨ref⟩ var! t₂)
            [ (λ w → ∃[ l₂ ] (⟦ w ≡ loc l₂ ⟧ ∗ l₂ ↦ t₂)) ]by⟨⟩
            [ (λ w → ∃[ l₂ ] (⟦ w ≡ loc l₂ ⟧ ∗ l₂ ↦ t₂) ∗ IsList t₂ ys ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]autorestore
            [ (λ w → ∃[ l₂ ] ((⟦ w ≡ loc l₂ ⟧ ∗ l₂ ↦ t₂) ∗ IsList t₂ ys ∗ l₁ ↦ t₁ ∗ IsList t₁ ys)) ]by⟨ ∗-exists ⟩
          ⟨in⟩
            [ _ ]exists⟨ l₂ ⟩
            [ ⟦ a ≡ loc l₂ ⟧ ∗ l₂ ↦ t₂ ∗ IsList t₂ ys ∗ l₁ ↦ t₁ ∗ IsList t₁ ys ]by⟨ ∗-assocᵣ ⟩
            [ l₂ ↦ t₂ ∗ IsList t₂ ys ∗ l₁ ↦ t₁ ∗ IsList t₁ ys ]pure⟨ a≡l₂ ⟩
            ⟨let _ =⟩
              [ Emp ]frameby⟨ ∗-idˡ ⟩
              (⟨fst⟩ var! v₂)
              [ (λ w → ⟦ w ≡ x ⟧ ∗ l₂ ↦ t₂ ∗ IsList t₂ ys ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]restoreby⟨ id ⟩
            ⟨in⟩
              [ l₂ ↦ t₂ ∗ IsList t₂ ys ∗ l₁ ↦ t₁ ∗ IsList t₁ ys ]pure⟨ x≡ ⟩
              [ Emp ]frameby⟨ ∗-idˡ ⟩
              (var! x ⟨cons⟩ var! (loc l₂))
              [ (λ w → ((⟦ w ≡ cons-val x (loc l₂) ⟧ ∗ l₂ ↦ t₂) ∗ IsList t₂ ys) ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]restoreby⟨ ∗-assocₗ ∘ ∗-assocₗ ⟩
              [ (λ w → (⟦ w ≡ cons-val x (loc l₂) ⟧ ∗ l₂ ↦ t₂ ∗ IsList t₂ ys) ∗ (l₁ ↦ t₁ ∗ IsList t₁ ys)) ]by⟨ ∗-monoᵣ ∗-assocᵣ ⟩
              [ (λ w → IsList w (x ::ˡ ys) ∗ l₁ ↦ t₁ ∗ IsList t₁ ys) ]by⟨ ∗-monoᵣ (λ { x@(emp refl •⟨ _ ⟩ _) → mkcons (∗-pure⁻ˡ x) }) ⟩
              [ (λ w → IsList w (x ::ˡ ys) ∗ IsList (cons-val x (loc l₁)) (x ::ˡ ys)) ]by⟨ ∗-monoₗ mkcons ⟩
              [ (λ w → IsList w xs ∗ IsList v xs) ]∎
            ⟨end⟩
          ⟨end⟩
        ⟨end⟩
      ⟨end⟩
    ⟨end⟩ })})})
  ⟨end⟩
[ (λ w → IsList w xs ∗ IsList v xs) ]∎
}
```

Listing 6.3: Mechanized Hoare style proof outline of copy list function in Agda.

Chapter 7

Discussion

The goal of this thesis is to write verified readable proof outlines for imperative programs in Agda. In the previous chapters I have presented a method on how to achieve this in a mechanization independent of target language, store implementation, location type and meta language. The meta language should be a higher order logic proof assistant or have support for dependent types, in order to define a shallow embedding for Separation Logic and a deep embedding of a target language. Furthermore, the meta language should support custom declarable syntax to declare syntax for triples. Finally, if the meta language has support for proof automation by reflection the outlines can be made more readable by implementing solvers for entailment. I demonstrated the idea by an implementation in the functional dependently typed programming language Agda for a simple imperative language. Agda has support for all of the above listed features. However, the implementation suffers from limitations of Agda and the mechanization itself. In this chapter we will discuss what the limitations are and how we can improve in future work.

7.1 Agda Limitations

The mechanized proof outlines are limited to the syntax of Agda. I found that Agda has two limitations in syntax when writing proof outlines with custom declared syntax.

Firstly, the declared syntax does not allow for pattern matching. So, a custom syntax for a lambda abstraction cannot pattern match on the argument of the lambda. The declared syntax for the EXISTS and PURE rule use this type. For example, using the PURE rule in an outline in Agda looks like:

```
[ ... ]pure⟨ P ⟩
...
```

Agda does not allow matching on the constructors of P declared syntax. For example, the `refl` constructor of propositional equality. It is possible, but then the outline has to be written with Agda pattern matching lambda:

```
[ ... ]pure (λ { refl →
...
})
```

This introduces two sets of braces that the user needs to match and that should not be necessary. If Agda had support for pattern matching inside declared syntax, some proof outlines could become more readable.

The second problem is that names or syntax in Agda cannot coincide with other already existing names or syntax. This is why the syntax of the language in the proof outlines is enclosed by angles. Otherwise, Agda will complain when writing programs and outlines because it does not know which syntax to parse. However, the types of the two syntax is different. The syntax for outlines is a triple, whereas the syntax for programs is of the language type. It seems that it might be possible to infer from the context in which the syntax is used which syntax is meant to be parsed.

A completely different approach that would not suffer from the limitations of the meta language is to design a Domain Specific Language (DSL) that extends the target language with triples and inference rules for proving specifications. A parser and kind of type checker can be implemented in a meta language that parses a program annotated with specifications and checks if the program satisfies the specifications and possibly solve unsolved specifications. This allows for any type of custom syntax, without the meta language having support for declared syntax. The interactivity of the proof assistant, however, is lost. Providing assistance and feedback to the user writing a proof is an essential part of writing complex proofs of programs. Feedback could be implemented through an IDE for the target programming language. A good example is Jahob by Zee, Kuncak, and Rinard (2009). Jahob is an integrated proof language for Java that allows programmers to annotate programs with specifications and dispatch proof obligations to an automated solver. The solution should also provide predicates for common data structures or allow for writing custom predicates or even derive predicates from data type declarations.

7.2 Features of the Language

The features of the example imperative language is not representative of modern imperative languages. It does not have support for types, input/output, loops, control flow breaking operations such as `continue` and `return`, pointer arithmetic, nondeterminism, garbage collection, exceptions, concurrency and more. While the example language demonstrates how to construct proof outlines for imperative programs inside Agda, the programs are far from practical. Writing proof outlines for these features would require designing new triples. Hoare Logic for these kind of features have been implemented before, but it remains to be seen how these would translate to readable proof outlines.

In this thesis we defined Hoare triples to be total, therefore, the proof outlines must be total. Programs must cover every path and be terminating. However, in many imperative programming languages this is not the case. Some programs are designed to not terminate, such as web servers, and some functions are partial and may, therefore, fail. For example, allocations may go wrong, because the requested amount of memory is not available. The specification for allocation presented in Chapter 4 assumes that allocation always succeeds. It is possible to make the total Hoare Logic partial through coinduction (Charguéraud 2020), which is also possible in Agda[1]. Partial triples specify that a program either runs indefinitely or correctly evaluates according to the specification. Failure can then be modeled as looping indefinitely.

7.3 Outlines in ANF

In Chapter 4 we defined the triples only on terms in Administrative Normal Form (ANF), therefore, the outlines are also in ANF. This greatly reduces the amount of programs we can write outlines for. Usually, imperative programs are not written in ANF, since it requires a lot of typing and thinking of new names to bind every expression to a name before you can use it. ANF, however, can be used as an intermediate representation for compilation of imperative programs. If a compiler already provides a transformation to ANF and it is

[1] `https://agda.readthedocs.io/en/v2.6.2/language/coinduction.html`

semantics preserving, then a user could write a program not in ANF and have the compiler transform it to a form that can have outlines. Because the transformation preserves semantics the original program also satisfies the specification by Lemma 4.2.1 that coerces triples for a term to another that has the same semantics. There is, however, still a disconnect from the original program and the program the user writes an outline for.

Another solution would be to design usable triples for every construct with arbitrary subexpressions, such that not only the triple for **let** has sequencing of state. The current triples make it difficult to reference the return value of a subexpression in the postcondition of the entire expression. This can either be solved by the same structure of the triple for **let** or by adding a pure fact on the equivalence of the return value. The latter can be formally stated as follows:

$$\textsc{alt-let-triple}\quad \frac{\{\eta \vdash H\}\, t_1\, \{w.\, [\![w = v]\!] * Q_1\} \qquad \{\{x \to v\} \cup \eta \vdash Q_1\, v\}\, t_2\, \{Q\}}{\{\eta \vdash H\}\, \texttt{let}\ x = t_1\ \texttt{in}\ t_2\, \{Q\}}$$

This structure forces the user to include a pure fact on every triple of a subexpression. Furthermore, you need a frame, which is not always there when reasoning with an empty state. Essentially, the rule takes care of matching on the equality for us. However, the current rule LET-TRIPLE is a more generic version and allows for any type of postcondition. In fact, ALT-LET-TRIPLE can be derived from LET-TRIPLE. Further work is necessary to develop triples for sequencing of subexpressions that can be used for writing mechanized proof outlines.

7.4 Proof Automation

In Chapter 5 we discussed a sound solver for entailment in Separation Logic given that two expressions had the same normal form. Furthermore, the solver was able to deduce an unknown frame inside an entailment. This leads to more concise mechanized proof outlines. However, the solver is limited in the set of entailments it can solve and the set of frames it can find. For example, the solver treats pure predicates and existential quantifiers as atomic expressions and can, therefore, not solve anything inside those predicates. Furthermore, the solver cannot remove and insert pure predicates, as we have seen in the swap example in Section 4.3, where we have to manually discard the return value of assignment. The problem with pure predicates in the solver is that the solver solves via a normal form. A pure predicate should be included in the normal form, but if one side of equivalence does not have the pure predicate then it has to be inserted. So, the solver should ask the user for an instance of the pure predicates so it can construct a pure predicate in the normal form. The existential quantifiers could also be treated as expressions to be able to solve expressions within quantifiers and eliminate variables and introduce them to the meta language like we do with the EXISTS rule. This would be useful since existential quantifiers are created when allocating new resources as we have seen in the copy list case study in 6.

There exist other tools for verification with Separation Logic that include tactics in Coq for rewriting expressions with pure predicates and existential quantifiers (McCreight 2009; Chlipala 2011; Cao et al. 2018; Charguéraud 2020). The other solutions use rewrite rules to simplify an expression, whereas the solver from this thesis uses a sound normal transform to prove the equivalence given that the normal forms are equal. The rewrite rules may not terminate, whereas the sound solver will always terminate. The rewrite rules, however, can solve a larger set of expression in SL. I chose to implement a sound solver, because it provides good feedback to the user why it is not possible to solve and it is safe. Rewriting is supported by Agda[2], but it is an unsafe feature of Agda, since it may break confluence of termination and type preservation (Cockx 2020). Moreover, the feature in Agda only supports propositional

[2]`https://agda.readthedocs.io/en/v2.6.2/language/rewriting.html`

equality, while we want to solve for entailment and equivalence of predicates. Further work is necessary to extend the solver for these cases in readable mechanized proof outlines. The solver from this thesis can be extended to support more expressions.

We can go even further with the amount of proof automation in proof outlines. Another useful feature would be to automate folding and unfolding of predicates, such as the IsList predicate from Chapter 6. This would require using propositions from the expression in finding an entailment. An example is folding an empty list:

$$[\![w = \texttt{nil}]\!] \vdash \text{IsList}(w, [])$$

The solver should be able to use the facts that w is a value of the empty list and **emp** entails IsList$(nil, [])$. This is similar to how automatic proof search works in Agda with Agsy[3] (Lindblad and Benke 2006). I, however, have not succeeded in using Agsy for proof automation for entailments in general. Smallfoot (Berdine, Calcagno, and O'Hearn 2005) is an example of a verification tool that can handle folding and unfolding list and tree specifications. Another proposal is to use automation to write the skeleton of proofs of programs. Since the outlines are based on an already existing program, the axioms and inference rules of the language to apply are fixed. So, after a user has written a program, they then have to write the program again but for the outline. It might be possible to write a macro that constructs an outline of the triples and leaves holes for frame and consequence. The macro might also try using the solver and only ask the user for proofs that it cannot solve by itself. This creates a more automated tool that would still be able to output readable proof outlines in the style of Hoare and also allow for interactive theorem proving.

Lastly, the proof automation implemented in Agda suffers from a performance issue. As already discussed in the case study in Section 6.3 solving entailments with more than five atomic variables consumes a lot of memory and does not terminate within a reasonable amount of time for interactively writing proofs where it is normal to get feedback within seconds and not minutes. The unification of the constructed terms and goal in the proof outline causes this performance issue. It is not certain what exactly causes it, but I was able to improve the performance by making the predicates abstract in Agda. Agda will not unfold abstract definitions. This improves performance, since Agda will not type check the definition of an abstract predicate during unification. However, this only increased performance minimally. Without abstract definitions it is possible to solve expressions with up to three atomic variables within reasonable time. When making the points-to relation abstract this number goes up to five. In proof outlines for more complex programs an expression could separate up to a dozen predicates. Further work is necessary to resolve these issues with macros and reflection in Agda. Furthermore, debugging and profiling these performance issues is tedious and incomplete in Agda and the errors are not always very helpful when writing macros, which made it difficult to analyze the problem. Adding and improving debug tools for Agda would make writing macros much easier.

If the above proposals could be implemented then the proof outline of `copy-list` from Chapter 6 could be just as concise as the manual proof outline. The advantage over the manual outline is that the mechanization also verifies the outline and assists the user in writing a correct proof.

[3]See for documentation on how to use proof automation in Agda: `https://agda.readthedocs.io/en/v2.6.2/tools/auto.html`

Chapter 8

Related work

This chapter relates this thesis to other scientific work in literature. Software verification tools can roughly be placed on a scale from interactive to fully automated. Interactive verification can be tedious work, but precise and strong specifications can be proven. This is illustrated by the extensive correctness proof of the cryptographic HMAC algorithm in openSSL by Beringer et al. (2015). On the other end we have automated tools that require minimal annotations and sometimes even no input from the user at all, but are constrained in the set of specifications they can assert. An example of which is the popular static analyzer Infer by Calcagno, Distefano, et al. (2015) developed at Meta that can statically analyze Java and C/C++ code for memory safety and more. However, proving more precise specifications, for example, about the contents of generic data structures, is not possible in fully automated tools. Most current verification tools are based on Separation Logic because the theory offers scalability (Pym, Spring, and O'Hearn 2019). The solution from this thesis is entirely interactive and leans a bit to automation. While there exist many tools that are more capable at verifying programs than the solution from this thesis, the mechanization structures proofs as proof outlines in the style of Hoare in Agda, which has not been done before. For a comprehensive survey of verification tools based on Separation Logic you may refer to the CACM article by O'Hearn (2019), especially the appendix, which lists many automated and semi-automated tools for program verification. In the remainder of this chapter we will look at other work from literature that implement tools for interactive program verification and compare them to the work from this thesis.

8.1 Interactive Verification in the style of Hoare

The work that perhaps most closely resembles the work from this thesis is the early tool called Cocktail by Franssen (2000). Cocktail is an interactive editing environment that combines programming and interactive Hoare style reasoning. The solution allowed programmers to program in a simple While language that featured loops and mutable variables. However, it did not allow for reasoning about pointers. Separation Logic had not been published yet at that time. Furthermore, it provided automated theorem proving by rewriting rules to reduce the amount of theorem proving the programmer had to carry out. Since the automation was carried out by rewrite rules, solving a theorem may not terminate, which is not the case when using a sound solver such as the one presented in this thesis. Nevertheless, Cocktail provided a usable interactive programming environment with support for holes, where the programmer received feedback about proof obligations and could write readable proof outlines in the style of Hoare. Unfortunately, I was not able to find any further development of Cocktail.

Another tool that comes close to reasoning about programs in the style of Hoare is Jahob by Zee, Kuncak, and Rinard (2009). It was an extension of Java that allowed programmers to add annotations to specify properties of the code. The tool mostly eliminated interactive theorem proving by dispatching proof obligations to solvers and using an integrated proof language where necessary. Since they created Jahob in a programming environment the programmer does not have to leave their trusted environment and due to the proof automation and custom interactive theorem proving the programmer does not have to have experience in a proof assistant. Jahob can also prove properties about more complex programs with object oriented programming in Java. However, Jahob is not based on Separation Logic and, therefore, the specifications are not as concise as the ones described by SL.

To my knowledge, more tools that allow reasoning about programs in the style of Hoare based on Separation Logic do not exist. Other mechanizations of Separation Logic for program verification in Agda are also lacking. However, there do exist verification tools that facilitate interactive reasoning about programs with tactics in the Coq proof assistant. Tactics are automated steps in a proof that reduce a goal to one or more smaller subgoals that need to be proven in order to generate a complete proof[1]. This is different than proof construction in Agda, because in our mechanization we have to explicitly construct a program—which corresponds to a proof—in order to prove a theorem.

8.2 Tactic Based Interactive Verification

Iris (Jung et al. 2018) is a higher-order concurrent Separation Logic framework that supports interactive program verification in the Coq proof assistant. The framework provides an "Iris Proof Mode" for Coq, which allows a user to interactively write proofs of program specifications using tactics(Krebbers, Timany, and Birkedal 2017). Similar to the solution presented in this thesis, Iris can be instantiated with a language defined by the user. Hoare triples in Iris are defined in the form of *weakest preconditions*. For a program t and postcondition Q, the weakest precondition of t and Q gives the *weakest* possible precondition for which t terminates and satisfies Q. In this context, weakest refers to the order on entailment, so every other precondition of a program can be shaped to the weakest. However, this style of specification is difficult to understand and does not profit from the same intuition as Hoare triples. Therefore, Iris also provides syntactic sugar to write Hoare triples in terms of weakest preconditions. Specifications for language constructs can then be defined in weakest precondition style similar to the specification in Hoare triples. This style of specifications is believed to be easier to automate than Hoare triples and is easier to use for sequencing specifications of subterms, which we have as a current problem in this thesis as seen in Chapter 7. Perhaps weakest precondition can be encoded in Agda and used as a backend of Hoare style proof outlines for easier automation. It would be important that the user can still reason in the layer of Hoare triples, since that more closely corresponds to program semantics.

Other excellent tools that mechanize Separation Logic in Coq are VST-Floyd presented by Cao et al. (2018), a framework for reasoning about concurrent C programs, and Bedrock by Chlipala (2011), a verification framework for reasoning about assembly level programs.

Finally, the verification for sequential programs by Charguéraud (2020) presents a mechanization of Separation Logic in Coq from the ground up for a simple sequential programming language similar to the one from this thesis. It also shows how to encode non-deterministic evaluation and partiality in a mechanization of Hoare Logic. The accompanying course by Charguéraud has inspired much of the work from this thesis[2]. The related work section also lists many mechanizations of Separation Logic in other proof assistants, except for Agda.

Although the tools presented in this section are very capable of proving complex diverse program specifications through automation, the proofs are not in the style of Hoare where

[1]For more information about tactics in Coq see: `https://coq.inria.fr/refman/proof-engine/tactics.html`

[2]The course can be found at: `https://softwarefoundations.cis.upenn.edu/slf-current/index.html`

the proof steps are clearly visible[3,4,5]. Progress is made in a proof through tactics and the easiest way to read the proof is to execute it in Coq, which points out the steps and subgoals. This is different from the mechanization presented in this thesis. The proof of a program also serves as a standalone proof that can be read without using or having knowledge of Agda.

[3]See the Iris example repository for proofs of programs: `https://gitlab.mpi-sws.org/iris/examples`

[4]See the VST repository for examples of proofs: `https://github.com/PrincetonUniversity/VST/tree/master/progs`

[5]See the Bedrock2 repository for examples of proofs: `https://github.com/mit-plv/bedrock2/tree/master/bedrock2/src/bedrock2Examples`

Chapter 9

Conclusion

This thesis presents a mechanization of Hoare style proof outlines that combines the readability of Hoare style proof outlines and the precision of mechanized tools. The solution is based on Separation Logic and is by design independent of target language and store implementation, which makes it flexible to use with other languages and memory models. The thesis also presents proof automation to simplify the process of writing mechanized proof outlines and make them more concise and readable. The solver can automatically reshape a restricted set of specifications in Separation Logic and automatically find a frame to temporarily reason with the minimal amount of resources. I demonstrated the proof outlines in the dependently typed programming language Agda on a simple imperative programming language based on λ-calculus. In a case study on a program that copies a list in memory, I showed that it is possible to write readable mechanized proof outlines in Agda.

Further research is necessary to explore readable mechanized proof outlines on more complex language features and specifications. If writing outlines inside a proof assistant becomes to restricted, we could design a DSL or IDE integration for languages to create an interactive environment for writing proof outlines while programming. Another point for further improvement is proof automation. The proof automation is not as advanced as other tools that provide deductive reasoning about programs, nonetheless, it shows how proof automation can be incorporated into more readable outlines. The solver can easily be extended to solve more complex steps in the outline and reduce the amount of details introduced by mechanization that are implicit in manual proof outlines. Examples include automatic folding and unfolding of predicates and introducing new variables. Currently, the outlines can only be written for programs in ANF, which is not usual for writing imperative programs. Further research is necessary to design mechanized outlines that allow for usable reasoning of programs not in ANF.

This thesis also serves as a case study for mechanization in Agda. The implementation provides a new mechanization of Separation Logic, Hoare triples and concrete store implementations that follow the laws of Separation Logic. The demonstration advertises the capabilities of Agda: declarable syntax, interactive theorem proving and proof automation through reflection. Although proof automation in Agda still shows room for improvement. The type checking of reflections is slow and quickly consumes a lot of memory even on small problem sizes. Furthermore, debugging and profiling macros is cumbersome and not automated. Nevertheless, the functional dependently typed programming language and proof assistant provides an excellent basis for writing complex mechanizations. I hope this work inspires more students, programmers and computer scientists to pursue formal verification and perhaps even in Agda.

Bibliography

Agda Development Team (2021). *Agda 2.6.2 documentation*. URL: https://agda.readthedocs.io/en/v2.6.2.

Berdine, Josh, Cristiano Calcagno, and Peter W. O'Hearn (2005). "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 115–137. DOI: 10.1007/11804192_6.

Beringer, Lennart et al. (2015). "Verified Correctness and Security of OpenSSL HMAC". In: *24th USENIX Security Symposium*, pp. 207–221. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer.

Birkedal, Lars, Noah Torp-Smith, and Hongseok Yang (2005). "Semantics of Separation-Logic Typing and Higher-Order Frame Rules". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. IEEE, pp. 260–269. DOI: 10.1109/LICS.2005.47.

— (2006). "Semantics of Separation-Logic Typing and Higher-Order Frame Rules for Algol-Like Languages". In: *Logical Methods in Computer Science* 2.5. DOI: 10.2168/LMCS-2(5:1)2006.

Bornat, Richard (2000). "Proving Pointer Programs in Hoare Logic". In: *Mathematics of Program Construction*. Ed. by Roland Backhouse and José Nuno Oliveira. Vol. 1837. Berlin, Heidelberg: Springer, pp. 102–126. DOI: 10.1007/10722010_8.

Boutin, Samuel (1997). "Using Reflection to Build Efficient and Certified Decision Procedures". In: *International Symposium on Theoretical Aspects of Computer Software*. Ed. by Martín Abadi and Takayasu Ito. Vol. 1281. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 515–529. DOI: 10.1007/BFb0014565.

Bove, Ana, Peter Dybjer, and Ulf Norell (2009). "A Brief Overview of Agda—A Functional Language with Dependent Types". In: *International Conference on Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

de Bruijn, Nicolaas Govert (1972). "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5, pp. 381–392. DOI: 10.1016/1385-7258(72)90034-0.

Burstall, Rodney M. (1972). "Some techniques for proving correctness of programs which alter data structures". In: *Machine intelligence* 7, pp. 23–50.

Buxton, John N. and Brian Randell (1970). *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969*. URL: https://dl.acm.org/doi/book/10.5555/1102021.

Calcagno, Cristiano, Dino Distefano, et al. (2015). "Moving Fast with Software Verification". In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi.

Vol. 9058. Lecture Notes in Computer Science. Cham: Springer, pp. 3–11. doi: 10.1007/978-3-319-17524-9_1.

Calcagno, Cristiano, Hongseok Yang, and Peter W. O'hearn (2001). "Computability and Complexity Results for a Spatial Assertion Language for Data Structures". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 2245. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 108–119. doi: 10.1007/3-540-45294-X_10.

Cao, Qinxiang et al. (2018). "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs". In: *Journal of Automated Reasoning* 61.1, pp. 367–422. doi: 10.1007/s10817-018-9457-5.

Carvalho, Marco et al. (2014). "Heartbleed 101". In: *IEEE Security & Privacy* 12.4, pp. 63–67. doi: 10.1109/MSP.2014.66.

Charguéraud, Arthur (2020). "Separation Logic for Sequential Programs (Functional Pearl)". In: *Proceedings of the ACM on Programming Languages* 4.ICFP, pp. 1–34. doi: 10.1145/3408998.

Chlipala, Adam (2011). "Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 46. PLDI '11. New York, NY, USA: Association for Computing Machinery, pp. 234–245. doi: 10.1145/1993498.1993526.

Cockx, Jesper (2020). "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules". In: *25th International Conference on Types for Proofs and Programs (TYPES 2019)*. Ed. by Marc Bezem and Assia Mahboubi. Vol. 175. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2:1–2:27. doi: 10.4230/LIPIcs.TYPES.2019.2.

Dijkstra, Edsger W. (1968). "A Constructive Approach to the Problem of Program Correctness". In: *BIT Numerical Mathematics* 8.3, pp. 174–186.

Floyd, Robert W. (1967). "Assigning Meanings to Programs". In: *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics*. Vol. 19. American Mathematical Society, pp. 19–32.

Franssen, M.G.J. (2000). "Cocktail: a tool for deriving correct programs". PhD thesis. Eindhoven University of Technology. doi: 10.6100/IR539431.

Grégoire, Benjamin and Assia Mahboubi (2005). "Proving Equalities in a Commutative Ring Done Right in Coq". In: *Theorem Proving in Higher Order Logics*. Ed. by Joe Hurd and Tom Melham. Vol. 3603. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 98–113. doi: 10.1007/11541868_7.

Hoare, C.A.R. (1969). "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10, pp. 576–580. doi: 10.1145/363235.363259.

Ishtiaq, Samin S. and Peter W. O'Hearn (2001). "BI as an Assertion Language for Mutable Data Structures". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 14–26. doi: 10.1145/360204.375719.

Jung, Ralf et al. (2018). "Iris from the ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic". In: *Journal of Functional Programming* 28, e20. doi: 10.1017/S0956796818000151.

Krasner, Herb (2018). *The Cost of Poor Software Quality in the US: A 2018 Report*. Tech. rep. Consortium for Information & Software Quality™ (CISQ™). url: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/index.htm.

— (2021). *The Cost of Poor Software Quality in the US: A 2020 Report*. Tech. rep. Consortium for Information & Software Quality™ (CISQ™). url: https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report.htm.

Krebbers, Robbert, Amin Timany, and Lars Birkedal (2017). "Interactive Proofs in Higher-Order Concurrent Separation Logic". In: *Proceedings of the 44th ACM SIGPLAN Symposium* *on Programming Languages*. POPL '17. New York, NY, USA: Association for Computing Machinery, pp. 205–217. DOI: 10.1145/3093333.3009855.

Leroy, Xavier (2021). *The CompCert C Verified Compiler: Documentation and User's Manual*. Tech. rep. Inria. URL: https://hal.inria.fr/hal-01091802.

Lindblad, Fredrik and Marcin Benke (2006). "A Tool for Automated Theorem Proving in Agda". In: *Types for Proofs of Programs*. Ed. by Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner. Vol. 3839. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 154–169. DOI: 10.1007/11617990_10.

Martin-Löf, Per (1982). "Constructive Mathematics and Computer Programming". In: *Logic, Methodology and Philosophy of Science VI*. Ed. by L. Jonathan Cohen et al. Vol. 104. Studies in Logic and the Foundations of Mathematics. Hannover: Elsevier, pp. 153–175. DOI: 10.1016/S0049-237X(09)70189-2.

McCreight, Andrew (2009). "Practical Tactics for Separation Logic". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 343–358. DOI: 10.1007/978-3-642-03359-9_24.

Norell, Ulf (2007). "Towards a practical programming language based on dependent type theory". PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology.

— (2008). "Dependently Typed Programming in Agda". In: *International School on Advanced Functional Programming*. Vol. 5832. Springer, pp. 230–266. DOI: 10.1007/978-3-642-04652-0_5.

O'Hearn, Peter W. (2019). "Separation Logic". In: *Communications of the ACM* 62.2, pp. 86–95. DOI: 10.1145/3211968.

O'Hearn, Peter W. and David J. Pym (1999). "The Logic of Bunched Implications". In: *Bulletin of Symbolic Logic* 5.2, pp. 215–244. DOI: 10.2307/421090.

O'Hearn, Peter W., John C. Reynolds, and Hongseok Yang (2001). "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 1–19. DOI: 10.1007/3-540-44802-0_1.

Parkinson, Matthew, Richard Bornat, and Cristiano Calcagno (2006). "Variables as Resource in Hoare Logics". In: *21st Annual IEEE Symposium on Logic in Computer Science*. IEEE, pp. 137–146. DOI: 10.1109/LICS.2006.52.

Pym, David J., Jonathan M. Spring, and Peter W. O'Hearn (2019). "Why Separation Logic Works". In: *Philosophy & Technology* 32.3, pp. 483–516. DOI: 10.1007/s13347-018-0312-8.

Reynolds, John C. (2002). "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

Rouvoet, Arjen (2021). "Correct by Construction Language Implementations". PhD thesis. Delft University of Technology.

Sabry, Amr and Matthias Felleisen (1992). "Reasoning about Programs in Continuation-Passing Style". In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP '92. New York, NY, USA: ACM, pp. 288–298. DOI: 10.1145/141478.141563.

Wadler, Philip (2015). "Propositions as Types". In: *Communications of the ACM* 58.12, pp. 75–84. DOI: 10.1145/2699407.

van der Walt, Paul and Wouter Swierstra (2012). "Engineering Proof by Reflection in Agda". In: *Implementation and Application of Functional Languages*. Ed. by Ralf Hinze. Vol. 8241. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 157–173. DOI: 10.1007/978-3-642-41582-1_10.

Zee, Karen, Viktor Kuncak, and Martin C. Rinard (2009). "An Integrated Proof Language for Imperative Programs". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. New York, NY, USA: Association for Computing Machinery, pp. 338–351. DOI: 10.1145/1542476.1542514.

Appendix A

Implementation

The mechanization in Agda can be found at:

`https://github.com/Olavhaasie/hoare-proof-outlines`

The implementation depends on and has been tested with the following:

- Agda v2.6.2:
  `https://github.com/agda/agda/tree/v2.6.2`
- Agda standard library v2.0 (commit hash `ae0702e`):
  `https://github.com/agda/agda-stdlib/tree/ae0702e`
- Ternary library (commit hash `fe131ce`):
  `https://github.com/ajrouvoet/ternary.agda/tree/fe131ce`

Appendix B

Proof Supplement

This appendix provides proof supplements of proofs that were left out for conciseness.

B.1 Soundness of Concatenation

Lemma B.1.1 (Concatenation of normal forms is sound)**.** Evaluating a concatenation of two normal forms n_1 and n_2 is equivalent to evaluating the normal forms separately.

$$\text{eval}_n(\text{concat}(n_1, n_2), \rho) \dashv\vdash \text{eval}_n(n_1, \rho) * \text{eval}_n(n_2, \rho)$$

Proof. We prove the equivalence by induction on the first normal form n_1. This results in showing the equivalence for a base case of **nil** and an induction case of **cons** $x\ n_1'$.
In the base case, when n_1 is **nil**, then the equivalence becomes:

$$\text{eval}_n(\text{concat}(\textbf{nil}, n_2), \rho) \dashv\vdash \text{eval}_n(\textbf{nil}, \rho) * \text{eval}_n(n_2, \rho)$$

Then, by definition of concat and eval$_n$ we can simplify the equivalence to:

$$\text{eval}_n(n_2, \rho) \dashv\vdash \textbf{emp} * \text{eval}_n(n_2, \rho)$$

This equivalence holds because **emp** is the identity of $*$ by Lemma 2.2.1, thus proving the base case.
In the induction case, when n_1 is **cons** $x\ n_1'$, then the equivalence becomes:

$$\text{eval}_n(\text{concat}(\textbf{cons}\ x\ n_1', n_2), \rho) \dashv\vdash \text{eval}_n(\textbf{cons}\ x\ n_1', \rho) * \text{eval}_n(n_2, \rho)$$

Then, by definition of concat and eval$_n$ we can simplify the left hand side of the equivalence to:

$$(\text{lookup}(x, \rho) * (\text{eval}_n(\text{concat}(n_1', n_2), \rho))$$

Then we apply the induction hypothesis on eval$_n$(concat(n_1', n_2), ρ) by monotonicity of $*$ to arrive at:

$$\text{lookup}(x, \rho) * (\text{eval}_n(n_1', \rho) * \text{eval}_n(n_2, \rho))$$

By associativity of $*$ we can reorder to:

$$(\text{lookup}(x, \rho) * \text{eval}_n(n_1', \rho)) * \text{eval}_n(n_2, \rho)$$

Finally, we can fold the lookup and eval$_n$ by definition of eval$_n$ to get:

$$\text{eval}_n(\textbf{cons}\ x\ n_1', \rho)) * \text{eval}_n(n_2, \rho)$$

which is equal to the right hand side of the equivalence and, thus, showing the IH.
Therefore, by the base case and IH, concatenation of normal forms is sound. □

B.2 Soundness of Flattening

This is a proof for Lemma 5.2.5.

Proof. We have to prove the following equivalence:

$$\text{eval}_n(\text{flatten}(H^{\downarrow}), \rho) \dashv\vdash \text{eval}(H^{\downarrow}, \rho)$$

We will prove using induction on the expression $H^{\downarrow}$. We have to show that the equivalence holds for the two base cases **emp**′ and x and the induction case of $x *' y$.
First, for the base case of **emp**′, the equivalence becomes:

$$\text{eval}_n(\text{flatten}(\mathbf{emp}'), \rho) \dashv\vdash \text{eval}(\mathbf{emp}', \rho)$$

Then, by definition of flatten and eval we can simplify to:

$$\mathbf{emp} \dashv\vdash \mathbf{emp}$$

which is equivalent by reflection of $\dashv\vdash$ (see Lemma 2.2.2). Therefore, proving the case for **emp**.
Second, for the base case of x, the equivalence becomes:

$$\text{eval}_n(\text{flatten}(x), \rho) \dashv\vdash \text{eval}(x, \rho)$$

Then, by definition of flatten and eval we can simplify to:

$$\text{lookup}(x, \rho) * \mathbf{emp} \dashv\vdash \text{lookup}(x, \rho)$$

which is equivalent, because **emp** is the identity of $*$ by Lemma 2.2.1. Therefore, proving the case for x.
Finally, for the induction case of $x *' y$, the equivalence becomes:

$$\text{eval}_n(\text{flatten}(x *' y), \rho) \dashv\vdash \text{eval}(x *' y, \rho)$$

We can unfold the left hand side by definition of flatten to:

$$\text{eval}_n(\text{concat}(\text{flatten}(x), \text{flatten}(y)), \rho)$$

By soundness of concatenation (see Lemma B.1.1) this is equivalent to:

$$\text{eval}_n(\text{flatten}(x), \rho) * \text{eval}_n(\text{flatten}(y), \rho)$$

Now we can apply the IH using monotonicity of $*$ (see Lemma 2.2.3) on both parts of the separation to arrive at:

$$\text{eval}(x, \rho) * \text{eval}(y, \rho)$$

which we can fold by definition of eval into:

$$\text{eval}(x *' y, \rho)$$

to arrive at our goal of showing the equivalence for the case $x *' y$.
Therefore, by showing that flattening is sound for all cases, we have proven that flattening an expression is sound. □

B.3 Soundness of Sorting

This is a proof for Lemma 5.2.6.

Proof. We will show that the following equivalence holds:

$$\text{eval}_n(\text{sort}(n), \rho) \dashv\vdash \text{eval}_n(n, \rho)$$

The sorted normal form sort(n) is a permutation of the normal form n, a reordering of the variables in the normal form. We can show that a normal form is equivalent to its reordering by $*$-commutativity, $*$-associativity and $*$-monotonicity. Therefore, evaluating a normal form is equivalent to evaluating the sorted normal form. □