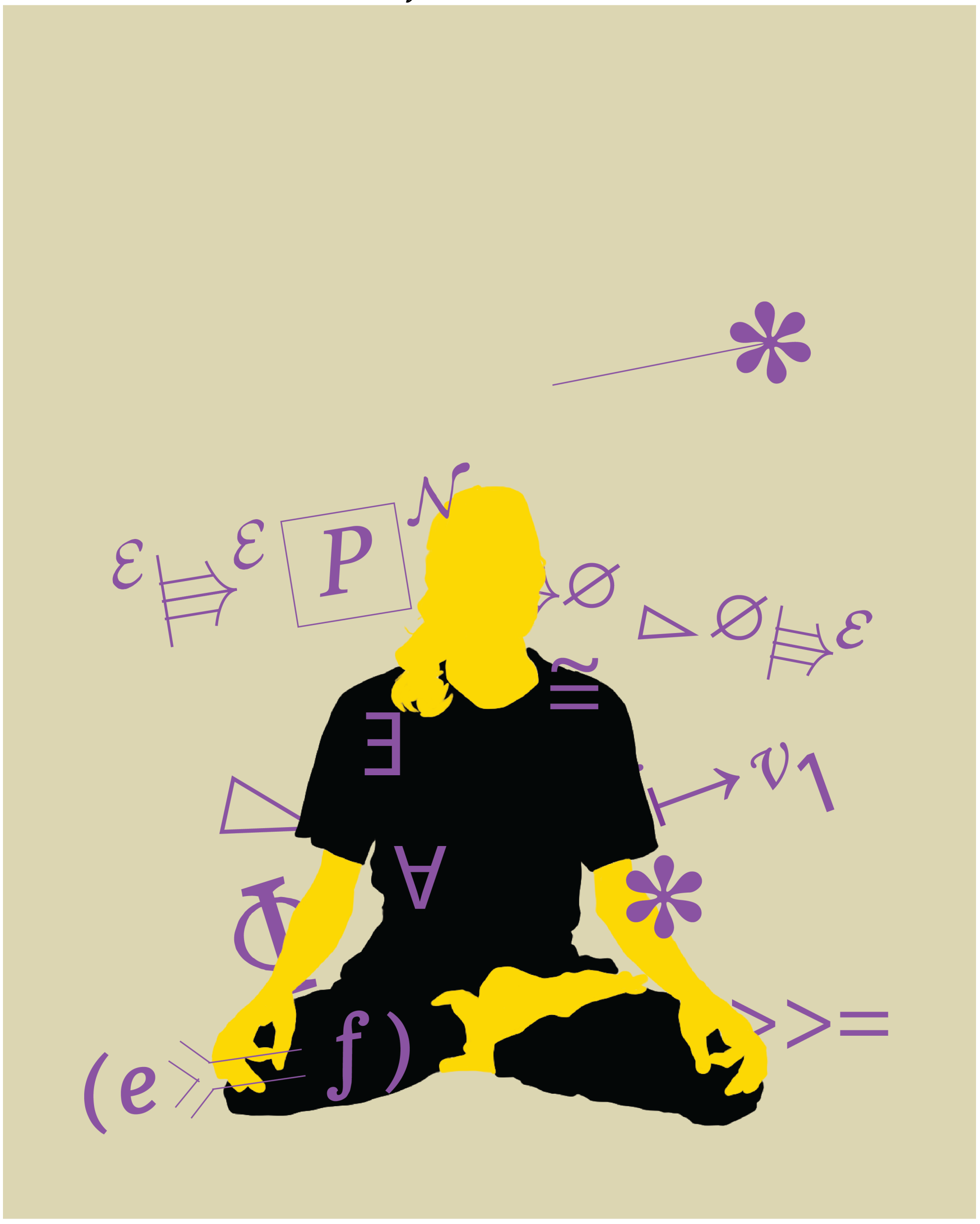
Proving functional correctness of monadic programs using separation logic

Version of December 2, 2022



Liam Clark

Proving functional correctness of monadic programs using separation logic

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Liam Clark
born in Amsterdam, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Proving functional correctness of monadic programs using separation logic

Author:	Liam Clark
Student id:	4303423
Email:	`lclark@student.tudelft.nl`

Abstract

Interaction trees are an active development in representing effectful and impure programs in the Coq proof assistant. Examples of programs they can represent are programs that use: mutable state, concurrency and general recursion. Besides representing these programs we also want to reason about and verify these programs using separation logic. That is the purpose of this thesis. More technically speaking interaction trees are new way to do shallow embeddings in the Coq proof assistant. They are a coinductive variant of the free monad and come with the usual constructions of events and event handlers. The aim of interaction trees is to represent impure programs and potentially non-terminating programs in their environment. Interaction trees are, in contrast to relational operational semantics, executable by interpretation or program extraction. Interaction trees come with a framework for reasoning about their behavior based on equivalency up to weak bisimulation. An open problem is to reason about interaction trees utilizing a separation logic rather than weak bisimulation. We developed Pothos as a solution to this problem. Pothos has an Iris based concurrent separation logic for interaction trees. We address the problem in a non-extensible setting, with mutable state, non-termination and concurrency as our chosen effects. Pothos inherits all the executable properties from interaction trees and includes a novel relation of Iris's step-index with coinductive types. We have proven our logic to be sound and include a case study of a spin lock library. The case study shows that our logic is both non-trivial and can utilize the standard Iris patterns for concurrency.

Preface

When I started with my computer science degree in Delft. It felt like coming home to something I was meant to do all along. I had a fierce passion for the field. Somewhere along the way, sadly, I had definitely lost my passion and that feeling. Working through this thesis, with ups and downs, through good times and through bad times, that has been replaced. A quiet contentment in its place. I am happy with the work I have done and I hope that anyone reading this may enjoy it.

I would like to deeply thank my supervisor Robbert Krebbers for his patience and guidance during this thesis project. I am thankful for all the time you gave to this project. I would like to thank the entire committee for their feedback. Most of all I would like to thank my father for giving me the opportunity to undergo my degree and for always believing in me. To wrap this up, I will give you my personal philosophy during this thesis:

> Even bad repetitions build towards good results

Liam Clark
Delft, the Netherlands
December 2, 2022

Contents

List of Figures

Chapter 1

Introduction

This thesis is a journey through the world of software verification. Software verification can be seen as a trade-off between trust, certainty and cost. We all have pieces of our code bases on which we want to enforce some degree of correctness, but how much are we willing to pay for that? Of course the exact same thing could be said about software testing. Testing and verification are two sides of the same coin. Software testing generally operates at a much lower level of cost and at the same time a much lower level of certainty. In software testing we run the code through a concrete set of values. We aim to pick a good representation of examples, to give us confidence that our code is correct for more general cases.

One view of verification is to prove that a program upholds a mathematical specification. Therefore verification is a two step process. We have to create a specification then show our program obeys it. Verification will generally give us more certainty at a much higher cost. Testing and verification also share common obstacles, a program that is difficult to test will likely be difficult to verify also. Specifically programs that contain any of the following effects: mutable state, concurrency and non-termination / non-determinsm.

Mutable state complicates the testing process, because we must first bring the program into the right state and ensure that we clean up all state (to not interfere with further test runs). When doing verification, mutable state can invalidate properties about memory that we have proven. Concurrency introduces non-determinsm into the execution order, which can introduce flakiness into a test suite. Concurrency also complicates verification, since specifications now need to hold for all possible execution orders. If we have non-termination we can get test cases that take forever, we either need to cut them off or decide they loop and remove them. Verifying programs with non-termination becomes more difficult because our logic now needs to deal with infinite executions. We cannot simply avoid these three effects, because they are essential for many applications. Instead we attempt to improve our verification tools to handle them.

For any verification system there are a lot of choices to make and the first choice is how much we want to interact with the verification system. There are fully automatic systems like Infer (Calcagno and Distefano 2011), systems that just require annotations on the source code like: Smallfoot (Berdine, Calcagno, and O'Hearn 2005), Verifast (Jacobs, Smans, and Piessens 2010), Viper (Müller, Schwerhoff, and Summers 2016) and Vercors (Blom and Huisman 2014) or fully interactive proof assistants. Examples of proof assistant are: Isabelle (Nipkow, Paulson, and Wenzel 2002) Agda (Norell 2008), F-star (Swamy, Hritcu, et al. 2016) and Coq (Coq-development-team 2000b). We will be using the Coq proof assistant in this thesis. Proof assistants in general are interactive programs that aid people in the construction of formal proofs. Proof assistants allow a user to pose their mathematical definitions and theorems, they then aid the user in proving their theorems by ensuring the proofs are correct and (usually) providing some means of proof automation. In general there is a trade-off between automatization and power between the before mentioned verification tools, as more input is given by the user stronger guarantees become achievable. Infer for example is fully automatic and can give a guarantee of memory safety, but not for all memory safe programs. The annotated automatic tools can offer functional correctness guarantees, but for a smaller class of programs than a fully interactive proof assistant could. The proof assistants give the largest possibilities for verifying full functional correctness of programs, but the downside is that they require a high level of precision to use (the before mentioned costs). Examples of these verification projects are: a fully verified C compiler CompCert (Leroy 2009), a fully verified microkernel sel4 (Klein et al. 2009) and a proof of the rust language safety claims RustBelt (Jung, Jourdan, et al. 2018).

To perform verification within a proof assistant we need a representation of our programming language and a program logic to reason about that language. To define our language inside our proof assistant we have the choice between shallow and deep embeddings (Wildmoser and Nipkow 2004) of our language. A shallow embedding shares features with the host language, that is in a shallow embedding we will directly use certain constructs of the proof assistant to define language constructs. Examples of this are: name binding and data types. Any record defined in Coq is immediately available as a value in our shallow embedding. A shallow embedding also comes with the added benefit that the type checker can check parts of our program. A deep embedding on the other hand gives an explicit model of the language's syntax and then gives an explicit semantics for this syntax. A deep embedding will give us more flexibility in defining our language semantics, since we are not tied to choices made by the host language. This flexibility comes at the cost of having to define much more ourselves. Shallow embeddings are often a monad (Wadler 1992). Monads were introduced to capture the semantics of computation and are therefore a popular choice for embeddings.

The challenge for a shallow embedding is to capture our before mentioned effects of: mutable state, concurrency and non-termination; inside the proof assistant. The Interaction trees (Xia et al. 2020) framework is an active development for creating shallow embeddings in the Coq proof assistant. To cite the authors own description: "Interaction trees" (itrees) are a general-purpose data structure for representing the behaviors of recursive programs that interact with their environments". Interaction trees take the form of a coinductive variant of the free-er monad (Kiselyov and Ishii 2015). Their coinductive nature is what allows them to model recursion and facilitate non-termination. Interaction trees have their own framework for reasoning about program refinement based on bisimilarity (which we discus in section 7.1), but an open challenge (and the goal of this thesis) is the development of Pothos: a separation logic for interaction trees.

There are many styles of verification, all with a rich tradition behind them. Since we are interested in programs with mutable state, a good starting point in history is the program logic: Hoare logic (Hoare 1969). Hoare logic was one of the first developments for the verification of stateful programs. Hoare logic is an inference system for programs and their pre- and postconditions, this means we have rules that link an atomic piece of source code to pre- and postconditions. We can then chain these rules together in a derivation to create proofs of specifications for entire programs. The way Hoare logic deals with state is by tracking changes in the state of the program inside of the pre- and postconditions. However when Hoare logic is applied to larger programs the information about the state of the program will blow up considerably. This is due to the fact that Hoare logic has a global view of the state, meaning we also have to specify state that is irrelevant to the procedure we are specifying. As our programs grow larger there is more and more irrelevant state that we have to specify.

Separation logic (O'Hearn, Reynolds, and Yang 2001; Reynolds 2002) was developed as an improvement on top of Hoare logic, to tackle exactly the problem of state blow up. The core idea about separation logic is to only have to specify the parts of the state that change, assuming everything else stays the same if it is not mentioned. This principle is called local reasoning and makes separation logic scale to projects such as Rustbelt (Jung, Jourdan, et al. 2018). Separation logic achieves local reasoning by making its statements aware of the heap and supporting the partitioning of the heap. The partitioning creates isolated pieces of heap that are easier to reason about. A split in the heap is created by the separating conjunction (written as $*$) and it is where the logic gets its name.

When Hoare logic and separation logic were developed for single threaded, sequential programs. The separation logic principle of local reasoning also works well for concurrent programs, which led to the development of concurrent separation logic (O'Hearn 2004; O'Hearn 2007). All the state that is non-shared can be reasoned about locally and does not have to be mentioned in the parts of the program that are concurrent. Only the concurrent parts have to deal with the difficulties of concurrency.

With interaction trees as our first building block we can now introduce our second building block, namely the concurrent separation logic we will be using: "Iris a framework for higher-order concurrent separation logic which has been implemented in the Coq proof assistant" (Jung, Krebbers, et al. 2018). Iris brings together a lot of sophisticated program verification techniques such as: Invariants, higher-order ghost state for named propositions, Löb induction for reasoning about recursion and higher-order logic for giving modular specifications. Iris is also step-indexed to make both Löb induction and higher-order ghost state possible. Iris is a stand alone logical framework providing building blocks for creating program logics, rather than a program logic for a specific language. This flexibility allows it to be used as the foundation of other verification projects such as: Perrenial a verifier for the go programming language (Chajed et al. 2019) and Actris a separation logic for message-passing programs (Hinrichsen, Bengtson, and Krebbers 2020). These applications of Iris are all done with deep embeddings. Iris also comes with its own deeply embedded language: Heaplang. The flexibility and verification features of Iris is what makes a suitable building block for Pothos.

There is one more trade-off to consider for Pothos, whether we want to build an intrinsic or an extrinsic logic. The trade-off is whether the program know about its specification or not. The first option is an intrinsic model, here the program stores the proof for its specification, therefore writing the program and verifying it become one single activity. On the other hand we have the extrinsic model, here we can first write the program, then verify it later. The big difference is the workflow these models give, with an intrinsic model we have to think about the specification as we write our program. This can make programming and verifying feel as a more united activity. In the extrinsic model we have the benefit that we can express our entire program, before we have to deal with its specification.

We can now properly define Pothos:

Pothos is an external separation logic for shallowly embedded programs using interaction trees and Iris.

Finally there are other developments that do full functional correctness utilizing separation logic such as: SteelCore (Swamy, Rastogi, et al. 2020), Iris and FCSL (Sergey, Nanevski, and Banerjee 2015). We can organize these projects by the design choices they made and place Pothos in their context.

	Intrinsic	Extrinsic
Shallow	SteelCore	Pothos / FCSL
Deep	$\times$	Iris

Contributions Our contributions are:

- We develop a shallowly embedded language based on interaction trees supporting: mutable state, non-termination and concurrency. For this language we build an external Iris based verification system. We do so to show that separation based reasoning is feasible for interaction trees with the effect environment limited to these three effects.
- At the same time we develop Iris based logics for the state and delay monad. These together with the logic for interaction trees display what Iris looks like in a shallow setting.
- We have a novel relation between step indexing and coinductive types to facilitate non-termination in a shallow embedding.
- We have formalized all of Pothos to prove our logics sound and adequate. The formalization of Pothos is available on Github (`https://github.com/LiamClark/weakest-pre`) and Zenodo (`https://doi.org/10.5281/zenodo.7339476`)
- We do a case study of a spin lock library and an application of that library to show that our logic is both non-trivial and can utilize the standard Iris patterns for concurrency.

This thesis is then structured in the following way:

Chapter 2 to 4 First a series of chapters that will progressively build up our expression language and logical system. We explore three monads: the state monad, the delay monad and interaction trees. Each monad has its own chapter. We will progressively incorporate the following language features:

- State. Since our goal is to create a separation logic for our expression language it must be able to express computations that have multiple mutable cells of state.
- Non-termination. We consider non-termination for two reasons. First it allows us to write algorithms that are not structurally recursive. Secondly non-termination is also required for implementing locking primitives. Namely it allows us to spin in an infinite loop whilst waiting to acquire a lock.
- Concurrency. Concurrency is the final goal for Pothos. We need an expression language that allows programmers to write concurrent programs. We need to do this in a manner that gives us usable semantics for the separation logic.

Each of these chapters follows a similar recipe: We give their definition and operations as Coq code. We give an example program in this expression language and present the high-level rules of our logic. Then we perform verification of the example program. Finally we present an adequacy theorem for the logic we developed in that chapter.

Chapter 5 In chapter 5 we cover the underlying semantics to our logics to explain how they are built on top of Iris, to improve confidence in the soundness of our logics and introduce the final Iris concepts required for chapter 6.

Chapter 6 chapter 6 is the final showcase of all that was built during this thesis. We verify a program with concurrent threads and spin-locks written in our interaction trees based language. To show our language is non-trivial and can use the standard Iris constructs for verification.

Chapter 2

State monad

Our first candidate is the state monad, modified to allow for failure. This chapter is structured in the following way: We first give the state monad definition and operation as Coq code (section 2.1). We then introduce the basics of separation logic (subsection 2.2.1). We use this separation logic to create a Hoare logic for our state monad (section 2.2). The Hoare logic will be used to verify a state monad example program (section 2.3). Then we repeat the last two steps but use weakest precondition rather than Hoare triples (section 2.4, section 2.5). Finally we give an adequacy theorem for our weakest preconditions (section 2.6).

2.1 The state monad definition

The definition we use is like the canonical state monad, but the result tuple is wrapped in the `option` type.

```
Record state (ST A: Type): Type :=
    State {
       runState: ST → option (A * ST)
    }.
```

It has an implementation of the monad type class, but we implement this as two separate type classes in Coq.

```
Instance mret_state ST: MRet (state ST) :=
  λ (A: Type) (a: A), State $ λ s, Some (a, s).

Instance mbind_state ST: MBind (state ST) :=
  λ (A: Type) (B: Type)
    (f: A → state ST B)
    (ma: state ST A), State $
                        λ st ⇒ match runState ma st with
                                | Some (x, st') ⇒ runState (f x) st'
                                | None ⇒ None
                                end.
```

The instance for `MRet` successfully returns the given value with the state unchanged. The `MBind` instance takes an input state `st`, then runs the first computation `ma` and extracts its result. If `ma` succeeded we can run the computation `f x`. So `MBind` creates a composite computation from `ma` and a continuation `f`. We introduce the standard notation for bind: `ma ≫= f` reads as `mbind f ma`.

We define two sets of operations on top of the state monad. The first is the usual set of operations for operating on a single cell of state in Figure 2.1. On top of that we introduce operations that can work with a heap of cells in Figure 2.2.

```
Definition getS {ST}: state ST ST :=
  State $ λ st, Some (st, st).

Definition putS {ST} (x: ST): state ST () :=
  State $ λ st, Some (tt, x).

Definition fail {ST A}: state ST A :=
  State $ λ st, None.

Definition ret_fail {ST A} (m: option A): state ST A :=
  match m with
  | Some x ⇒ mret x
  | None ⇒ fail
  end.
```

Figure 2.1: Single cell operations

First are the two standard operations for get and put (here `getS` and `putS`). Then we have two non-standard operations. The `fail` operation exposes unconditional failure. Lastly the `ret_fail` operation allows us to lift a value that may have failed (`option A`) into our monad, either returning a value if it was present or failing if there was not. Now we have the heap operations:

```
Definition modifyS' {A} (n: nat)
  (f: gmap nat A → gmap nat A): state (gmap nat A) () :=
    State $ λ st, if decide (is_Some (st !! n)) then Some (tt, f st) else None.

Definition get {A} (n: nat): state (gmap nat A) A :=
  getS ≫= λ st, ret_fail $ lookup n st.

Definition put {A} (n: nat) (x : A) : state (gmap nat A) unit :=
  modifyS' n  <[n := x]>.

Definition alloc (v: V) : state V A nat :=
  modifyS $ λ st,
            let l := fresh $ dom (gset nat) st
            in (l, <[l:= v]> st).

Definition free {A} (n: nat): state (gmap nat A) unit :=
 modifyS' n (delete n).
```

Figure 2.2: Heap operations

Our heaps are finite partial functions from natural numbers to values of type: `A`. For the representation we use the type `gmap nat A` from stdpp (Iris-development-team 2022), with natural numbers as adresses and values of generic type `A`. All the heap operations work in the following way: first retrieve the entire heap from the state monad, then perform an operation on a single cell in the heap. The `modifyS'` function is specifically important since it makes operations on empty cells fail. The `get` operation looks up a single cell by address in the heap. The `put` operation modifies one cell by updating an entry at a certain address. The `alloc` operation finds the first free address in the heap and inserts a value at this address, returning the address that was selected. The `free` operations deletes a cell at the specified address. Note that `put` and `free` both fail if the location was not present in the map before hand, meaning we can only write and free previously allocated locations.

2.2 State monad Hoare logic

Now with our state monad established we can focus on the verification side. We first present a minimal subset of the Iris (Jung, Krebbers, et al. 2018) separation logic. Then we present a Hoare logic (Hoare 1969) for our state monad. Finally we show an equivalent system using weakest preconditions.

2.2.1 Introduction to separation logic

In this section we give a intuition for separation logic as it is present in Iris. Iris introduces iProp as the type of propositions to represent our separation assertions. An iProp can be seen as a predicate over a heap. An example of this is the points-to connective $l \mapsto v : \mathsf{iProp}$. This describes a heap which must contain a location l that points to a value v, it may also contain other locations. The main connective regarding heaps is the separating conjunction which is written as $P * Q$. The idea is that $*$ separates the heap into two disjoint parts, meaning if we have ownership over a location that ownership is exclusive. Lastly there is the separating implication or wand, written as $P \mathrel{-\!\!*} Q$, this can be read as we can give up P to obtain Q. A big part of Iris is left out here, namely that an iProp is also step-indexed. We elaborate the extra features of Iris in Chapter 3. We now present the basic separation logic rules:

$$\mathsf{True} * P \dashv\vdash P \qquad P * Q \vdash Q * P \qquad (P * Q) * R \vdash P * (Q * R) \qquad \overset{*\text{-MONO}}{\dfrac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}}$$

$$\overset{-\!*\text{-INTRO}}{\dfrac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}} \qquad \overset{-\!*\text{-ELIM}}{\dfrac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}}$$

The first rule states that a proposition proves itself and can have True added or removed. The next two rules state that $*$ is commutative and associative. Then we have the $*$-MONO rule which allows us to split our context to prove both sub-propositions of the $*$ separately. Finally we have the rules for $-\!*$. the introduction should be familiar, but note that Q is introduced with the $*$ instead of the normal conjunction $\wedge$. Finally we have our elimination rule for $-\!*$ which can be used to derive the more standard elimination rule $(P * (P \mathrel{-\!\!*} Q) \vdash Q)$ as such:

$$\dfrac{\dfrac{\dfrac{}{P \mathrel{-\!\!*} Q \vdash P \mathrel{-\!\!*} Q}}{(P \mathrel{-\!\!*} Q) * P \vdash Q}\ -\!*\text{-ELIM}}{P * (P \mathrel{-\!\!*} Q) \vdash Q}\ *\text{-COMMUTATIVE}$$

With our basic separation logic in place we will now establish a Hoare logic on top of it in the next section.

2.2.2 Hoare logic basics

The main connective of Hoare logic is the Hoare triple. A Hoare triple consists of a precondition for a program, a program and a postcondition for that program. We write a Hoare triple as such:

$$\{P\}\, e\, \{w.Q\}$$

Here e is a program and P is the precondition and Q is the postcondition. This can be read as: if P holds before we run e then e is safe and after e has run, Q will hold. More specifically P and Q are iProps. The Hoare triple itself is also an iProp and its type is:

```
iProp → state (gmap nat S) A → (A → iProp) → iProp
```

We now give the standard rules for separation logic based Hoare triples adapted for state monad based language in Figure 2.3:

$$\textsc{Hoare-s-bind}\quad \frac{\{P\}\, e\, \{Q\} \qquad \forall x, \{Q\, x\}\, f\, x\, \{R\}}{\{P\}\, e \gg\!= f\, \{R\}} \qquad\qquad \textsc{Hoare-s-seq}\quad \frac{\{P\}\, e_1\, \{w.Q\} \qquad \{Q\}\, e_2\, \{R\}}{\{P\}\, e_1\, ;;\, e_2\, \{R\}}$$

$$\textsc{Hoare-s-ret}\quad \{\mathsf{True}\}\, (\texttt{mret}\ x)\, \{w.w = x\} \qquad\qquad \textsc{Hoare-s-get}\quad \{l \mapsto v\}\, (\texttt{get}\ l)\, \{w.w = v * l \mapsto v\}$$

$$\textsc{Hoare-s-put}\quad \{l \mapsto v\}\, (\texttt{put}\ l\ v')\, \{w.w = () * l \mapsto v'\} \qquad\qquad \textsc{Hoare-s-alloc}\quad \{\mathsf{True}\}\, (\texttt{alloc}\ v)\, \{l.l \mapsto v\}$$

$$\textsc{Hoare-s-free}\quad \{l \mapsto v\}\, (\texttt{free}\ l)\, \{w.w = ()\} \qquad\qquad \textsc{Hoare-s-frame}\quad \frac{\{P\}\, e\, \{w.Q\}}{\{P * R\}\, e\, \{w.Q * R\}}$$

Figure 2.3: State monad Hoare logic

First we have the Hoare-s-bind rule that states if we have a Hoare triple for the continuation f for which the precondition matches the postcondition of our program e we can compose the triples. The Hoare-s-seq follows from Hoare-s-bind automatically by discarding the result of e_1. The rule Hoare-s-ret states that if we return a value, our program terminates with that value. Finally we have rules for the four heap operations:

- The rule Hoare-s-get states that if we own a cell at location l we can read it and obtain the value contained at l.

- The rule Hoare-s-put states that if we own a cell at location l we are allowed to update that cell to any value of our choosing. This makes sense since the points to connective gives us exclusive ownership.

- The rule Hoare-s-alloc states we can allocate a value on the heap and obtain its address as the return value.

- The rule Hoare-s-free states that if we own a cell of the heap, we can remove that value from the heap.

2.3 State monad Hoare logic verification

In this section we apply the Hoare logic we established on a program by giving a proof outline. We will consider the following program:

```
Definition prog_swap (l k: nat): state (gmap nat nat) unit :=
  x ← get l ;
  y ← get k ;
  put l y ;; put k x.
```

or equivalently

```
Definition prog_swap' (l k: nat): state (gmap nat nat) unit :=
  get l ≫= λ x,
    get k ≫= λ y,
      put l y ≫= λ _, put k x.
```

For this program we want to prove the following specification:

$$\{l \mapsto v * k \mapsto w\}\ \texttt{prog\_swap}\ l\ k\ \{l \mapsto w * k \mapsto v\}$$

Our precondition states that we own two locations l and k that point to values v and w respectively. Then in the postcondition we still own the same two locations, but the values pointed to have been swapped around.

We give an outline the proof for this program. Note that in this proof outline we implicitly apply the HOARE-S-BIND rule after every line:

```
Definition prog_swap (l k: nat): state (gmap nat nat) unit :=
  $\{l \mapsto v * k \mapsto w\}$
  x ← get l ;    (HOARE-S-GET)
  $\{l \mapsto v * k \mapsto w * (x = v)\}$
  y ← get k ;    (HOARE-S-GET)
  $\{l \mapsto v * k \mapsto w * (x = v) * (y = w)\}$
  put l y ;;    (HOARE-S-PUT)
  $\{l \mapsto w * k \mapsto w * (x = v) * (y = w)\}$
  put k x.    (HOARE-S-PUT)
  $\{l \mapsto w * k \mapsto v\}$
```

2.4 State monad weakest precondition

Having seen the Hoare logic for the state monad in action we now present the less intuitive, but more flexible weakest preconditions formulation of the same inference rules. We will use this style for the remainder of the document. The weakest precondition connective is written as such: $\mathsf{wps}\ e\ \{\Phi\}$, here e is a program and Φ a postcondition. The intuitive semantics behind the weakest precondition connective is that it is the weakest precondition, for which e is safe and e terminates with Φ. It is like taking a Hoare triple and removing its precondition. The removal of the precondition is also reflected in the type of the weakest precondition connective: `state (gmap nat S) A → (A → iProp) → iProp`. We can make the connection to the Hoare triple more concrete by relating the two constructs as follows [1]:

$$\{P\}\, e\, \{Q\} \triangleq P \vdash \mathsf{wps}\, e\, \{Q\}$$

Here we have recovered the full Hoare triple by means of entailment. We have a valid Hoare triple if the precondition P, entails the precondition constructed by the weakest precondition connective. We now give the inference rules for the weakest precondition connective in Figure 2.4:

$$\mathsf{wps}\, e\, \{x.\, \mathsf{wps}\, (f\, x)\, \{\Phi\}\} \vdash \mathsf{wps}\, (e \gg= f)\, \{\Phi\} \qquad (\textsc{wp-s-bind})$$
$$\Phi\, x \vdash \mathsf{wps}\, (\mathtt{mret}\; x)\, \{\Phi\} \qquad (\textsc{wp-s-ret})$$
$$l \mapsto v * (l \mapsto v \twoheadrightarrow \Phi\, v) \vdash \mathsf{wps}\, (\mathtt{get}\; l)\, \{\Phi\} \qquad (\textsc{wp-s-get})$$
$$l \mapsto v * (l \mapsto v' \twoheadrightarrow \Phi\, ()) \vdash \mathsf{wps}\, (\mathtt{put}\; l\; v')\, \{\Phi\} \qquad (\textsc{wp-s-put})$$
$$\forall l.\, l \mapsto v \twoheadrightarrow \Phi\, l \vdash \mathsf{wps}\, (\mathtt{alloc}\; v)\, \{\Phi\} \qquad (\textsc{wp-s-alloc})$$
$$l \mapsto v * \Phi\, () \vdash \mathsf{wps}\, (\mathtt{free}\; l)\, \{\Phi\} \qquad (\textsc{wp-s-free})$$

Figure 2.4: State monad weakest preconditions

All the wp-rules presented here had a corresponding Hoare rule, therefore we explain how to read the rules compared to their Hoare counterpart. The first change is that we swapped from bar judgments towards turnstile judgments. Secondly the precondition has moved from the Hoare triple to the left side of the turnstile. This is the reason we now use the turnstile judgement, we need the Iris context to contain our precondition.

The last difference is that all the postconditions are left abstract and set to Φ. We do this because it makes the application of these inference rules easier when we do proofs with them. Let us look at the example of the wp-s-get rule. In the Hoare version we had this postcondition: $w.w = v * l \mapsto v$ in the weakest pre version it becomes: $l \mapsto v \twoheadrightarrow \Phi\; v$ We can see that any requirements on the heap, that we had in the Hoare version, are used to imply the postcondition Φ. The requirements of the return value are present in the argument given to Φ, here namely $\Phi\; v$. A similar translation is done for all the other rules.

Lastly we no longer require a frame rule on the level of weakest preconditions like with Hoare-s-frame, the Iris context takes care of this on its own.

2.5 State monad weakest precondition verification

We now want to verify the same program swap as seen in section 2.3. First we need to change the specification from a Hoare triple to a weakest precondition. We can take the precondition and simply use the wand to connect it to the weakest precondition as such: $(l \mapsto v * k \mapsto w) \twoheadrightarrow \mathsf{wps}\, (\mathtt{swap}\;\; l\;\; k)\, \{l \mapsto w * k \mapsto v\}$. In our derivation we will already have introduced the precondition into the context. Before we can give the whole derivation tree we establish a few shorthands for readability. The first being the way we manipulate the branching of the wp-s-get and wp-s-put rules. We call these the resource-shuffle for get and put:

[1]This definition restricts the nesting of Hoare triples because they themselves are no longer iProps. Enabling that nesting requires an extra modality (the persistence modality $\boxdot$) to make it sound. See "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" (Jung, Krebbers, et al. 2018) for the proper definition.

$$\dfrac{\dfrac{\dfrac{}{l \mapsto v \vdash l \mapsto v} \qquad \dfrac{P * l \mapsto v \vdash \Phi\ v}{P \vdash l \mapsto v \mathrel{-\!\!*} \Phi\ v}\ \textsc{-*-intro}}{l \mapsto v * P \vdash l \mapsto v * (l \mapsto v \mathrel{-\!\!*} \Phi\ v)}\ \textsc{*-mono}}{l \mapsto v * P \vdash \mathsf{wps}\ (\mathtt{get}\ l)\ \{\Phi\}}\ \textsc{wp-s-get}$$

$$\dfrac{\dfrac{\dfrac{}{l \mapsto v \vdash l \mapsto v} \qquad \dfrac{P * l \mapsto v' \vdash \Phi\ ()}{P \vdash l \mapsto v' \mathrel{-\!\!*} \Phi\ ()}\ \textsc{-*-intro}}{l \mapsto v * P \vdash l \mapsto v * (l \mapsto v' \mathrel{-\!\!*} \Phi\ ())}\ \textsc{*-mono}}{l \mapsto v * P \vdash \mathsf{wps}\ (\mathtt{put}\ \ l\ \ v')\ \{\Phi\}}\ \textsc{wp-s-put}$$

The point of the shorthand is to eliminate the branching going when proving we have access to the location for a WP-S-GET, WP-S-PUT or WP-S-FREE rule. Secondly we can optimize the reduction of the post condition. The predicates in the post condition are actually the same functions as the functions in our programs. For example we often have programs with the following shape: wps (`get l >>= λx, f x`) $\{\Phi\}$. An operation with a continuation formed with the bind operator. We then apply the bind rule which introduces a binder in the post condition. The binder is then applied to the lambda in the program: wps (`get l`) $\{z.$wps (`(λx, f x)` z) $\{\Phi\}\}$. This can be simplified by performing the function application in the program of the post condition. We then obtain: wps (`get l`) $\{z.$wps (`f` z) $\{\Phi\}\}$. At the end of the resource-shuffle we can see a value being applied to the entire post condition. This is what fills in our binder z, with the value actually on the heap v. We then unify this to obtain our final expression: wps (`f` v) $\{\Phi\}$. In the following derivations these simplifications will be done in the same fashion. Finally for the longer lines in the derivation tree we shorten the post condition to: post $\triangleq l \mapsto w * k \mapsto v$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{l \mapsto w * k \mapsto v \vdash l \mapsto w * k \mapsto v}}{l \mapsto w * k \mapsto v \vdash \mathsf{post}}\ \textsc{unfold post}}{k \mapsto w * l \mapsto w \vdash \mathsf{wps}\ (\mathtt{put}\ \ k\ \ v)\ \{\mathsf{post}\}}\ \textsc{resource-shuffle-put}}{l \mapsto v * k \mapsto w \vdash \mathsf{wps}\ (\mathtt{put}\ \ l\ \ w)\ \{\_.\mathsf{wps}\ (\mathtt{put}\ \ k\ \ v)\ \{\mathsf{post}\}\}}\ \textsc{resource-shuffle-put}}{\begin{array}{c} l \mapsto v * k \mapsto w \vdash \\ \mathsf{wps}\ (\mathtt{put}\ \ l\ \ w\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \_,\ \mathtt{put}\ \ k\ \ v)\ \{\mathsf{post}\}\end{array}}\ \textsc{wp-s-bind}}{\begin{array}{c} k \mapsto w * l \mapsto v \vdash \\ \mathsf{wps}\ (\mathtt{get}\ \ k)\ \{z.\mathsf{wps}\ (\mathtt{put}\ \ l\ \ z\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \_,\ \mathtt{put}\ \ k\ \ v)\ \{\mathsf{post}\}\}\end{array}}\ \textsc{resource-shuffle-get}}{\begin{array}{c} k \mapsto w * l \mapsto v \vdash \\ \mathsf{wps}\ (\mathtt{get}\ \ k\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \mathtt{y},\ \mathtt{put}\ \ l\ \ \mathtt{y}\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \_,\ \mathtt{put}\ \ k\ \ v)\ \{\mathsf{post}\}\end{array}}\ \textsc{wp-s-bind}}{\begin{array}{c} l \mapsto v * k \mapsto w \vdash \\ \mathsf{wps}\ (\mathtt{get}\ \ l)\ \{z.\mathsf{wps}\ (\mathtt{get}\ \ k\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \mathtt{y},\ \mathtt{put}\ \ l\ \ \mathtt{y}\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \_,\ \mathtt{put}\ \ k\ \ z)\ \{\mathsf{post}\}\}\end{array}}\ \textsc{rsrc-shuffle-get}}{\begin{array}{c} l \mapsto v * k \mapsto w \vdash \\ \mathsf{wps}\ (\mathtt{get}\ \ l\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \mathtt{x},\ \mathtt{get}\ \ k\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \mathtt{y},\ \mathtt{put}\ \ l\ \ \mathtt{y}\ \ \mathtt{>\!\!>\!=}\ \ \lambda\ \_,\ \mathtt{put}\ \ k\ \ \mathtt{x})\ \{\mathsf{post}\}\end{array}}\ \textsc{wp-s-bind}$$

2.6 State monad adequacy

Throughout this chapter we have seen programs in the state monad and verified these programs. We have not executed any programs, nor have we related our verification to the execution of our programs. This is exactly what we will address now.

Running state monad programs is quite straightforward, since the state monad is a record wrapper around a function. To run a state monad program, we take the function out of the record and call it. Given a program e of type `state (gmap nat S) A` and an initial heap σ of type `gmap nat S` then we can run our program as follows:

$$\texttt{runState}\ e\ \sigma$$

We can now state the following adequacy theorem:

Theorem 1 *Let Ψ be a first-order predicate (rather than a separation logic assertion). If* $\mathsf{True} \vdash \mathsf{wps}\ e\ \{\Psi\}$ *then there exists a value x and a heap st' such that:* $\texttt{runState}\ e\ st = \texttt{Some (x, st')}$ *and $\Psi\ x$ hold.*

Note the fact that having a valid weakest precondition is sufficient to imply that the program terminates with a `Some` (successfully), therefore we have total-correctness. The predicate Ψ is not a separation logic assertion but a first-order predicate, this means (a) Ψ can not contain anything regarding our heap and (b) that we can see Ψ as specification of our program in a regular logic (outside of the Iris context), for example a Coq `Prop`.

We have now learned all there is about the state monad.

Chapter 3

Delay monad

In this chapter we explore a monad for our second requirement: non-termination. We will be looking at the delay monad (Capretta 2005). We first give its definition (section 3.1) and show what programs can be written in the delay monad. Then we want to give weakest preconditions inference rules to verify these programs (section 3.3). The new rules require an extension of the separation logic constructs we have seen so far, therefore we show those first (section 3.2). We then show the verification of a program that never terminates (section 3.4). Finally we give an adequacy statement for our weakest preconditions (section 3.5).

3.1 Delay monad definition

The delay monad is a coinductive type. Formally an inductive type is the least fixpoint of a recursive type equation, whereas a coinductive type is the greatest fixpoint of a recursive type equation. This means that a value of a coinductive type can have an infinite amount of nested constructors. The definition of the delay monad is:

```
CoInductive delay (A: Type): Type :=
| Answer: A → delay A
| Think: delay A → delay A.
```

We distinguish two different classes of values of the delay monad. We can have a finite number of `Thinks` ending in an `Answer`. This value represents a computation that succeeds after some amount of computational steps. The other option is an infinite amount of `Thinks`, representing a divergent computation. We can express corecursive computations using Coq's Cofixpoints, here we can make corecursive calls without upholding the termination checker as long as we uphold Coq's guardedness condition (Coq-development-team 2000a). The guardedness condition exists to avoid logical inconsistencies. The condition requires every corecursive invocation to be wrapped in a constructor of the coinductive type that we are returning. For the delay monad the idea is as follows: if we make a recursive call, then we wrap the result in a `Think` constructor. In the base case we return our final result packaged in the `Answer` constructor. Coincidentally the `Answer` constructor matches the type signature of `mret`. We can see this machinery in action while we show the monad instance for `delay`.

```
Definition bind_delay {A B} (f: A → delay B): delay A → delay B :=
  cofix go (ma : delay A): delay B :=
    match ma with
    | Answer x ⇒ f x
    | Think ma' ⇒ Think (go ma')
    end.
```

```
Instance mret_delay : MRet delay := λ _ x, Answer x.

Instance mbind_delay : MBind delay :=
  λ _ _ f ma, bind_delay f ma.
```

We can reason about `bind_delay` for terminating and divergent computations. In the terminating case we apply our continuation `f` to the answer at the bottom of the stack of `Think` nodes. Then wrap an equal amount of `Think` nodes around the result of the continuation. If the value `ma` is a divergent computation, then `bind_delay` it self diverges. We endlessly spin in the `Think` branch.

We can also check `bind_delay` to see if it upholds our guardedness condition. We can see this happen in the `Think` case of `bind_delay`. Here we want to obtain a value of type `delay B` by recursively converting the sub-computation `ma'`. This is allowed because it is surrounded by the `Think` constructor.

In `bind_delay` there is an inner cofix, rather than having the top level definition as a CoFixpoint. This is required to make callers of `bind_delay` pass the guardedness condition, for example the function `iter` in the next section. The reason this matters is that: Coq will first try and normalize a definition before checking for guards and having more arguments fixed means Coq can normalize more.

There is a problem with the delay monad displayed by Chlipala (2013). The following program is not accepted by Coq:

```
CoFixpoint fib (n: nat): delay nat :=
  match n with
  | 0 ⇒ Answer 1
  | 1 ⇒ Answer 1
  | _ ⇒ n1 ← fib (pred n) ;
        n2 ← fib (pred (pred n)) ;
        Answer (n1 + n2)
  end.
```

The problem is the guardedness condition, namely we defined bind as a function and not a constructor, therefore it does not guard a recursive call. If bind is not a guard then we can not use the result of a recursive call to make another. The work of Xia et al. (2020) is a sophisticated solution to writing corecursive programs without restrictions of the guardedness condition, which we will see in chapter 4. For now we borrow a few of their iteration combinators and show them in the simpler `delay` setting.

Delay monad combinators

The first combinator is the `iter` combinator:

```
CoFixpoint iter {A B} (f: A → delay (A + B)) : A → delay B :=
  λ x, ab ← f x ;
       match ab with
       | inl a ⇒ Think (iter f a)
       | inr b ⇒ Answer b
       end.
```

We have a function `f` which represents a loop body that we want to execute multiple times. The function `f` returns a coproduct of `A` or `B`, using this `f` can signal whether its computation is done by yielding a `B`, or needs another iteration by yielding an `A`. We start the loop with an initial state of type `A` which is passed into the first execution of `f`. We match on the result of f, if f requires another iteration we run it again with the result of type A the prior execution yielded. This gives us a functional style of state passing between iterations.

We can run an extra iteration of the function f by recursing iter and guarding it with the Think constructor. If the function f returns a value of type B our computation is done.

The benefit of the iter combinator is that it handles all the guardedness concerns for any loop we want to write. The iter looping combinator is quite the natural fit once we add our state effect into the mix in chapter 4. It would be nicer to have access to full general recursion, but a looping combinator is a fair comprise. We now give the example of computing the Fibonacci sequence in the delay monad:

```
(* fib' is our loop body.
   st is the state that we use between iterations of our computation.
   its components are: (n, x, y) where
   n is the number of Fibonacci numbers we still need to create,
   once we reach 0 we yield our result.
   x is the current Fibonacci number
   y is the next Fibonacci number *)
Definition fib' (st: nat * nat * nat): delay ((nat * nat * nat) + nat) :=
  match st with
  |(0, x, y) ⇒ Answer $ inr $ x
  |(S n, x, y) ⇒ Answer $ inl (n, y, x + y)
  end.

(* applying our loop combinator to the loop body, then call it with an initial state *)
Definition fib (n: nat): delay nat := delaystate.iter fib' (n, 0, 1).
```

The example here follows the Fibonacci with an accumulator style, or can be reminiscent of an imperative implementation. There are a few delay specific elements that we will explain. First we match on the loop condition n. The zero branch returns an inr with our final answer. Recall from the definition of iter that an inr breaks the loop and yields the final result. The S n branch instead returns an inl where our state gets updated for the next iteration of the loop. The next iteration will have n one lower and our Fibonacci numbers shifted up in the sequence. Finally we need to return a value in the delay monad. Because this is required by the type signature of iter, however no recursive computation steps are taken in our loop body. Thus we can simply return our computation with the Answer constructor. Leaving all the necessary Think constructors up to iter.

The second looping combinator is called loop its definition is as follows:

```
Definition loop {A B C} (f: C + A → delay (C + B)): A → delay B :=
  λ a,
   iter (λ ca: C + A,
          f ca ≫= λ cb: C + B,
                   match cb with
                   | inl c ⇒ Answer $ inl $ inl c
                   | inr b ⇒ Answer $ inr b
                   end
        )
        (inr a).
```

The loop combinator gives us a bit more flexibility in what state is passed between loop iterations. This is achieved with the new type parameter C. The state between iterations does not have to be of type A, but can instead be of type C which is only present in the loop body `f`. The `iter` and `loop` combinators are interderivable and equally expressive. The `loop` combinator is included here mostly for completeness and from now on we will prefer the `iter` combinator.

3.2 The later modality

Before we can present a weakest preconditions calculus for the delay monad, we need to expand our separation logic. We add the later modality (written $\triangleright$) (Nakano 2000; Appel et al. 2007) which is commonly used in Iris-based logics to reason about general recursive functions. In subsection 2.2.1 we gave the intuition for iProp as predicates over heaps, we have to expand this for the later modality. The new intuition is that an iProp is a predicate over a heap and a natural number representing the current step of computation (heap → nat → Prop). If P is an iProp then $\triangleright P$ will hold at the next step of computation. The later modality advances us through these steps of computation, requiring the iProp to hold at the next step of computation. We now present its rules:

$$\begin{array}{cccc}
\text{▷-DIST} & \text{LÖB} & \text{▷-MONO} & \text{▷-INTRO} \\
\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q & \dfrac{\triangleright P \vdash P}{\mathsf{True} \vdash P} & \dfrac{P \vdash Q}{\triangleright P \vdash \triangleright Q} & P \vdash \triangleright P
\end{array}$$

Later-dist This rule states that the $\triangleright$ modality distributes over the separating conjunction $(*)$.

Löb-induction The Löb rule is our main tool to reason about potentially non-terminating programs. It states that to prove any P we can gain a hypothesis of $\triangleright P$ to prove it. This is logically consistent because of the $\triangleright$ modality guarding P. The main application of this rule is reasoning about our loop combinators. We use the Löb rule to gain an induction hypothesis about the next iteration of the loop. While reasoning about the first iteration of the loop we will proceed to the next step of computation. This will allow us to remove the $\triangleright$ guarding our induction hypothesis using the next rule.

Later-mono The ▷-mono rule states that if both the antecedent and consequent are guarded by a $\triangleright$ then we can remove it from both sides. Intuitively this make sense, if everything holds at the next step of computation, we can just go there and drop the $\triangleright$ guard. Note that this rule is the only way to remove a later guard from any of our hypotheses.

Later-intro The ▷-intro rule states that we are free to advance our consequent to next step of computation.

We will now present our new weakest preconditions rules that will utilize the later modality.

3.3 Delay monad weakest preconditions inference rules

With our new expression language in place we now present its corresponding weakest preconditions calculus in Figure 3.1. The type of our weakest precondition connective is: `delay A → (A → iProp) → iProp`. The main difference here is that our expressions no longer contain any state, thus all the rules regarding heap operations are no longer applicable. The rules for the heap operations will be reintroduced in chapter 4. We now cover the new rules.

$$\begin{array}{rcll} \mathsf{wpd}\ e\ \{x.\ \mathsf{wpd}\ (f\ x)\ \{\Phi\}\} & \vdash & \mathsf{wpd}\ (e \gg\!= f)\ \{\Phi\} & \textsc{wp-d-bind} \\ \Phi\ x & \vdash & \mathsf{wpd}\ (\mathtt{mret}\ x)\ \{\Phi\} & \textsc{wp-d-ret} \\ \triangleright\, \mathsf{wpd}\ e\ \{\Phi\} & \vdash & \mathsf{wpd}\ (\mathtt{Think}\ \ e)\ \{\Phi\} & \textsc{wp-d-think} \end{array}$$

$$\textsc{wp-d-iter}$$

$$\mathsf{wpd}\ (f\ x) \left\{ w.\ \begin{bmatrix} \triangleright\, \mathsf{wpd}\ (\mathtt{iter}\ f\ x')\ \{\Phi\} & \text{if } w = \mathtt{inl}\ x' \\ \Phi\ y & \text{if } w = \mathtt{inr}\ y \end{bmatrix} \right\} \vdash \mathsf{wpd}\ (\mathtt{iter}\ f\ x)\ \{\Phi\}$$

Figure 3.1: Delay monad weakest preconditions

Wp-think The wp-d-think rule shows how we tie the step indexes of Iris to the delay monad. Namely we see every `Think` constructor as one step of computation. This means that every corecursive call or loop iteration we make, is one step of computation. The way that wp-d-think states this is that we can remove an outer `Think` constructor, by proving the weakest precondition for the inner program, at the next step of computation.

Wp-iter The wp-d-iter rule has four parts.

1. In the consequent we can see that we are reasoning about the entire loop. The post condition should then hold when the loop is completely done.

2. In the antecedent in the outer weakest precondition we see a single execution of $f\ x$. Here we are reasoning about a single loop iteration. What happens here is that we split the loop in its first iteration and the rest of the loop.

3. The first part of our postcondition is if the loop execution ended up returning an `inl` x'. If we recall our definition of the iter combinator, this means the loop body requests another iteration. Here we do something similar to the wp-d-bind rule, we nest a weakest precondition that needs to hold for the rest of the loop. However that will be at our next step of computation, because to go to the next iteration we will always step through a `Think` constructor by definition of iter. Normally we discharge this nested weakest precondition by means of an induction hypothesis obtained from Löb induction. The later that is in front of the weakest precondition will allows to strip the later guard from the induction hypothesis by means of $\triangleright$-mono.

4. The second part of our postcondition is if the loop returned an `inr` y, this means the loop is done terminating with final answer y. This means we show that the postcondition Φ holds for y.

3.4 Delay monad verification

In this section we will show an application of the weakest pre rules of the delay monad by verifying the following program:

```
Definition loop_body {A}: unit → delay A :=
  delaystate.iter (λ x, mret $ inl ()).
```

```
Definition loop_prog {A}: delay A :=
    loop_body ().
```

This program simply loops forever, which should make sense if we recall the definition of `iter`. The `iter` function requests another iteration for an `inl`. We always return an `inl` hence we have an infinite loop.

We give it the following specification:

$$\vdash \text{wpd}\ \texttt{loop\_prog}\ \{\_.\text{False}\}$$

If we have a specification with the postcondition set to false, then that means we are proving the program never terminates. If the program would terminate, then the postcondition should hold. The postcondition is false and will never hold, hence the program should not terminate if this specification is provable. The equivalent Hoare triple for our specification would be: $\{\text{True}\}\ \texttt{loop\_prog}\ \{\_.\text{False}\}$. We now give the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\text{wpd}\ \texttt{iter (mret \$ inl ()) ()}\ \{\_.\text{False}\} \vdash \\ \text{wpd}\ \texttt{iter (mret \$ inl ()) ()}\ \{\_.\text{False}\}\end{array}}}{\text{wpd}\ \texttt{loop\_prog}\ \{\_.\text{False}\} \vdash \text{wpd}\ (\texttt{loop\_body ()})\ \{\_.\text{False}\}}\ \textsc{unfold-program}}{\triangleright\,\text{wpd}\ \texttt{loop\_prog}\ \{\_.\text{False}\} \vdash \triangleright\,\text{wpd}\ (\texttt{loop\_body ()})\ \{\_.\text{False}\}}\ \triangleright\textsc{-mono}}{\begin{array}{c}\triangleright\,\text{wpd}\ \texttt{loop\_prog}\ \{\_.\text{False}\} \vdash \\ \text{wpd}\ (\texttt{mret \$ inl ()})\left\{w.\ \left[\begin{array}{ll}\triangleright\,\text{wpd}\ (\texttt{loop\_body}\ x)\ \{\_.\text{False}\} & \text{if } w = \texttt{inl}\ x \\ (\_.\text{False})\ y & \text{if } w = \texttt{inr}\ y\end{array}\right]\right\}\end{array}}\ \textsc{wp-d-ret}}{\triangleright\,\text{wpd}\ \texttt{loop\_prog}\ \{\_.\text{False}\} \vdash \text{wpd}\ \texttt{loop\_prog}\ \{\_.\text{False}\}}\ \textsc{wp-d-iter}}{\vdash \text{wpd}\ \texttt{loop\_prog}\ \{\_.\text{False}\}}\ \textsc{Löb}$$

The first step in the proof is to use the Löb rule, to obtain an induction hypothesis to reason about the final iterations of the loop. Secondly we use the wp-d-iter rule to split the loop into its first iteration and its final iterations. We know the loop body does nothing except request the next loop iteration by returning an `inl ()`. We can propagate this information to the postcondition using the wp-d-ret rule, this picks the `inl` branch of the postcondition introduced by the wp-d-iter rule. We then need to show that the further iterations of the loop still do nothing, but at the next step of computation. This is exactly our induction hypothesis. We could finish the proof here, but we demonstrate the usage of the ▷-mono rule to show how to strip the later modality from our induction hypothesis. Removing the later modality from the induction hypothesis is often required in other proofs.

3.5 Delay monad adequacy

Running a delay monad program is slightly more complicated than a state monad program. We want to obtain the value in the `Answer` constructor (if there is one). To reach the `Answer` constructor we have to peel of a possibly infinite number of `Think` constructors. One way of dealing with this is by creating a fuel-based evaluator (Siek 2012; Owens et al. 2016) for the delay monad, which reports a failure if it did not reach an answer before running out of fuel. We define the evaluator as follows:

```
Fixpoint eval_delay {A} (n: nat) (ma: delay A): option A :=
  match n with
  | O ⇒ None
  | S n' ⇒ match ma with
           | Answer x ⇒ Some x
```

```
            | Think ma' ⇒ eval_delay n' ma'
            end
  end.
```

First we match on `n` to check if we still have fuel remaining, if not the evaluation terminates with a failure. If we do have fuel left over we pattern match on `ma` and if it is an `Answer` then evaluation terminates with that value. In the case of a `Think` constructor we make a recursive call with our fuel `n` decreased. The evaluation now always terminates, because it is structurally recursive on `n`.

We can now start formulating our adequacy theorem for the delay monad. The adequacy theorem for the state monad from section 2.6 was a total-correctness statement. Now that we have introduced the option of non-termination we change to partial-correctness.

Theorem 2 *Let Ψ be a first-order predicate (rather than a separation logic assertion). If* `True` $\vdash$ `wpd` $e\,\{\Psi\}$ *and* `eval_delay` $n\ e =$ `Some x` *then* $\Psi\ x$ *holds.*

In contrast to the previous adequacy theorem our adequacy theorem for the delay monad gives us partial-correctness. We have to do this because we now allow for never terminating programs. For the state monad having a valid weakest precondition was enough to ensure termination. For the delay monad this is simply not true as we have seen in Section 3.4. There we had a valid weakest precondition for a program that looped forever. If a program never terminates there is very little we can say about its postcondition. Therefore we add the requirement that the program terminates as our second assumption, giving us partial-correctness.

With the delay monad thoroughly explored we can now turn our attention to interaction trees.

Chapter 4

Interaction trees

In this chapter we consider interaction trees (Xia et al. 2020) as our third and final candidate for our expression language. The authors relate interaction trees to the free-er monad (Kiselyov and Ishii 2015), namely Interaction trees can be seen as a coinductive variant of the free-er monad. Interaction trees can utilize a similar technique as described in "Turing-Completeness Totally Free" (McBride 2015). Interaction trees help us achieve all three of our desired characteristics, namely they give us:

1. A way to express non-terminating computations through their coinductive nature.

2. A way to interpret concurrent computations because they can be suspended naturally.

3. A way to include effectful operations for state and concurrency.

This chapter is structured in the following way: We first give the definition of interaction trees and their operations as Coq code (section 4.1). We choose the environment we will use for our expression language (subsection 4.1.2). We then give a weakest precondition for interaction trees and show some example programs (section 4.2). We show an outline of the interpreter we use for interaction trees (section 4.3). Finally we state our adequacy theorem for interaction trees (section 4.4).

4.1 Interaction trees definition

Interaction trees are defined as follows [1] :

```
CoInductive itree (E: Type → Type) (R: Type): Type :=
  | Answer: R → itree E R
  | Think: itree E R → itree E R
  | Fork: itree E () →  itree E R → itree E R
  | Vis: forall {A: Type}, E A → (A → itree E R) → itree E R.
```

The first thing to notice is the fact that the type is defined as a `CoInductive` type. The type `itree` has two type variables. The type variable `E` represents the environment of available effects and the type variable `R` represent the type of the value this `itree` computes. We now cover all the constructors:

[1]There are three differences between the definition given here and that of Xia et al. (2020). First there is the inclusion of the `Fork` constructor. Secondly the `Answer` and `Think` constructors are renamed to make the names consistent with their delay monad counterparts. Finally the definition we use is always defined by constructors (the positive form), while Xia et al. (2020) eventually switches to a definition using destructors (the negative form).

Answer The `Answer` constructor is there to return a value, it is exactly the same as in the free-er monad and the delay monad from chapter 3.

Think The `Think` constructor comes directly from the delay monad (Capretta 2005). It allows for non-terminating computations without triggering any effects.

Fork The `Fork` constructor takes two arguments. The first argument is a sub-expression that returns the unit value. This will be the thread that gets forked-off, hence we do not want it to return a meaningful value. The second argument is a sub-expression with which the main thread continues, this does return a meaningful value. Technically speaking the `Fork` constructor could have been defined as part of the environment `envE` in subsection 4.1.2, leaving our definition of interaction trees closer to the original definition. However when we include `Fork` as a constructor, then all recursive occurrences of `itree` are now present in a single inductive type. Having all the recursive occurrences in a single type allows us to avoid doing tricky nested induction / coinduction in proofs.

Vis The `Vis` constructor takes three arguments, a type variable `A`, a command `e` and a continuation `k`. The command `e` is an effect to execute in the environment, for example mutating a heap cell. The command `e` has type: `E A`. The type variable `A` represents the type of value that the execution of the effect returns. For example if we execute an effect to retrieve a heap cell of type `nat` for that command `A` will become `nat`. This way the return value of the effect can depend on the value of the command executed.

The continuation `k` resumes the computation with the result of the effect. The type of `k` is: `A → itree E R`, it takes in the result of the effect (of type `A`) and returns the rest of the computation (of type `itree E R`). One way of thinking about this is as follows: We are building a computation with effects `E` and return type `R` (also known as `itree E R`). In the case of the `Vis` constructor we do this by executing an effect `E` with return value `A`. We then convert the return value of type `A` into the desired type `itree E R` through the extra computation represented by `k`.

Finally we need to adjust our intuition of when a computational step is taken. In chapter 3 the `Think` constructors took a step, which is still the case. Additionally, the `Fork` and `Vis` constructors will also perform a step.

4.1.1 Generic interaction trees operations

In this section we cover operations on interaction trees that work for any environment `E`. We first show that interaction trees form a monad. Then we present and expand upon the iteration combinators from chapter 3. We give the monad instances for interaction trees:

```
Instance itree_bind {E}: MBind (itree E) :=
  λ (R S: Type) (k: R → itree E S),
    cofix go (u : itree E R): itree E S :=
      match u with
      | Answer r ⇒ k r
      | Think e ⇒ Think (go e)
      | Fork ef e ⇒ Fork ef (go e)
      | Vis cmd k ⇒ Vis cmd (λ x, go (k x))
      end.

Instance itree_mret {E}: MRet (itree E) :=
 λ _ x, Answer x.
```

The monad instance for `itree` works similarly to the monad instance for the delay monad. The intuition for this monad instance is that our continuation `k` gets applied throughout the tree to the value in an `Answer` node if it exists. In code this means we want to convert every occurrence of the type variable `R` with `S` by means of `k`. In the `Answer` branch this means applying `k` to the value `r`, to convert it. In the other two branches it means corecursively converting the entire subtree. To convert the subtree we have our cofixpoint `go` of type `itree E R → itree E S`.

The `Think` branch is still straightforward, we apply `go` to the nested `itree` and satisfy the guardedness condition by wrapping the result back into a `Think` constructor.

A very similar thing happens for the `Fork` branch. We can safely ignore the forked-off thread and apply `go` to the nested `itree`.

In the `Vis` branch we still want to apply `go` to the nested subtree, but the subtree takes on a different form. The subtree is formed by the continuation `k` stored in the `Vis` constructor, therefore we do not have a value of `itree E R` to convert until we execute the continuation. The solution is to create a new `Vis` node with a continuation that applies `go` to the result of the original continuation `k`. Since we are creating a new `Vis` node the guardedness condition is also satisfied.

Finally we have the iteration combinators as given by Xia et al.:

```
CoFixpoint iter {E A B} (f: A → itree E (A + B)) : A → itree E B :=
  λ a, ab ← f a ;
       match ab with
       | inl a ⇒ Think (iter f a)
       | inr b ⇒ Answer b
       end.

Definition loop {E A B C} (f: (C + A) → itree E (C + B)): A → itree E B :=
  λ a,
    iter (λ ca,
            cb ← f ca ;
            match cb with
            | inl c ⇒ mret (inl (inl c))
            | inr b ⇒ mret (inr b)
            end
         )
         (inr a).
```

Both of these are exactly identical to the ones presented in chapter 3. There is a third recursion operator that is possible on interaction trees which will be discussed in chapter 7.

4.1.2 The environment

To have interaction trees work as our expression language we need to consider the environment we want to use them in. Working with an abstract or open environment brings along complications when creating a weakest precondition for interaction trees. We work around this by choosing a fixed environment. We discuss the complications in chapter 7, for now we want two things from this environment: mutable state and concurrency. Therefore we add an operation for each of the heap operations from chapter 2. For concurrency we require two things: A way to start new threads and some form of communication between threads. We already have the first with our `Fork` constructor, thus we add an operation for compare and exchange (closely related to the perhaps more familiar compare and swap) and all our requirements are met.

We give the entire definition of the environment we will be using:

```
Definition loc := nat.

Variant envE (V : Type): Type → Type :=
|GetE:      loc → envE V V
|PutE:      loc → V → envE V ()
|AllocE:    V → envE V loc
|FreeE:     loc → envE V ()
|CmpXchgE:  loc → V → V → envE V (V * bool)
|FailE:     forall R, envE V R.
```

Our type `envE` has a type parameter `V` for the type of values on the heap and a type index to indicate the return value of the command. Let us recall that we want to use `envE` for the type variable `E` of type `Type → Type` in `itree`. For the types to match we have to fix the type of `V`. We can now define our expressions to be: interaction trees modified with a fork primitive in the closed `envE` environment supporting values of type `V`.

```
Definition expr (V: Type) := itree (envE V).
```

We explain each of the constructors of `envE` in order:

Get The `GetE` constructor takes a `loc` as input, corresponding to the heap location it will look up the value with. Then in the return type we see that it specifies that if a `GetE` effect is performed it will return a value of type `V`.

Put The `PutE` constructor takes a `loc` and a new value to store at the specified `loc`. When a `PutE` operation is performed it returns a value of unit type.

Alloc The `AllocE` constructor takes a value of type `V` to allocate in a new memory location. When a `AllocE` operation is performed it returns the `loc` at which the value was allocated.

Free The `FreeE` constructor takes a `loc` to specify what cell to free. When a `FreeE` operation is performed it returns the unit type after freeing the cell.

Fail The `FailE` constructor takes no arguments and can return any value to the computation. There is now way to actually implement this in our evaluator, meaning that after seeing a `FailE` the entire computation stops. The promise of any return value will always be unmet.

Compare and exchange The `CmpXchgE` constructor is there to atomically compare and exchange a value with a value on the heap. The `CmpXchgE` constructor takes three arguments: A `loc` l to specify at what heap location we want to perform this swap and two values of type `V` which we refer to as v_1 and v_2. The compare and exchange operation then proceeds as follows: It then retrieves the value v_h at location l, compares if v_h is equal to v_1 and if it is equal store v_2 at l. Finally the operation returns `(vh, true)`. In the case that v_h was not equal to v_1 the operation stores nothing and just returns `(vh, false)`. We return the value v_h so the caller can try again using v_h as the new v_1. The boolean in the return value is there so that the caller can tell whether the heap was successfully updated.

We now have the problem that calling these operations is rather involved. First we need to create a `Vis` node then pass in the operation and manually give the continuation. To make this simpler we define a helper function for all our operations:

```
Definition get {V} (l: loc): expr V V          := Vis (GetE l) Answer.
Definition put {V} (l: loc) (v: V): expr V () := Vis (PutE l v) Answer .
Definition alloc {V} (v: V): expr V loc       := Vis (AllocE v) Answer.
Definition free {V} (l: loc): expr V ()        := Vis (FreeE l) Answer.
Definition fork {V} (e: expr V ()): expr V () := Fork e (Answer ()).
Definition fail {V R}: expr V R                := Vis FailE Answer.
Definition cmpXchg {V} (l: loc) (v1 v2: V): expr V (V * bool) :=
   Vis (CmpXchgE l v1 v2) Answer.
```

All these functions encapsulate the creation of the `Vis` constructor and use `Answer` as the continuation. This gives us the freedom to write the actual continuation using the bind operator.

4.2 Interaction trees weakest preconditions

In this section we present the weakest precondition inference rules for reasoning about single threaded `itree` programs. These will be expanded in chapter 5 to include the rules for multithreaded programs. The rules should essentially be the union of rules from chapter 2 and chapter 3 with new additions for `fork` and `cmpXchg` and can be found in Figure 4.1. There is no rule for the `fail` operation, because the `fail` operation crashes our program, therefore would always violate the safety property of our weakest precondition connective. Ultimately this means any program that might call fail, is not verifiable.

We introduce a new weakest precondition connective for `itrees`: $\mathsf{wpi}\ e\ \{\Phi\}$. We give a simplified type for the connective that works in a single-threaded setting of type: `expr V R → (R → iProp) → iProp`

$$
\begin{array}{rcll}
\mathsf{wpi}\ e\ \{x.\,\mathsf{wpi}\ (f\ x)\ \{\Phi\}\} & \vdash & \mathsf{wpi}\ (e \gg\!= f)\ \{\Phi\} & \textsc{wp-i-bind} \\
\Phi\ x & \vdash & \mathsf{wpi}\ (\texttt{mret}\ x)\ \{\Phi\} & \textsc{wp-i-ret} \\
\triangleright \mathsf{wpi}\ e\ \{\Phi\} & \vdash & \mathsf{wpi}\ (\texttt{Think}\ \ e)\ \{\Phi\} & \textsc{wp-i-think} \\
\triangleright l \mapsto v * \triangleright(l \mapsto v \twoheadrightarrow \Phi\ v) & \vdash & \mathsf{wpi}\ (\texttt{get}\ l)\ \{\Phi\} & \textsc{wp-i-get} \\
\triangleright l \mapsto v * \triangleright(l \mapsto v' \twoheadrightarrow \Phi\ ()) & \vdash & \mathsf{wpi}\ (\texttt{put}\ l\ v')\ \{\Phi\} & \textsc{wp-i-put} \\
\triangleright(\forall l.\, l \mapsto v \twoheadrightarrow \Phi\ l) & \vdash & \mathsf{wpi}\ (\texttt{alloc}\ v)\ \{\Phi\} & \textsc{wp-i-alloc} \\
\triangleright l \mapsto v * \Phi\ () & \vdash & \mathsf{wpi}\ (\texttt{free}\ l)\ \{\Phi\} & \textsc{wp-i-free} \\
\triangleright \mathsf{wpi}\ e\ \{\_.\mathsf{True}\} * \Phi\ () & \vdash & \mathsf{wpi}\ (\texttt{fork}\ e)\ \{\Phi\} & \textsc{wp-i-fork} \\
\triangleright l \mapsto v_1 * \triangleright(l \mapsto v_2 \twoheadrightarrow \Phi\ (v_1, \mathsf{True})) & \vdash & \mathsf{wpi}\ (\texttt{cmpXchg}\ l\ v_1\ v_2)\ \{\Phi\} & \textsc{wp-i-cmpXchg-suc} \\
v_1 \neq v_3 * \triangleright l \mapsto v_1 * & & & \\
\triangleright(l \mapsto v_1 \twoheadrightarrow \Phi\ (v_1, \mathsf{False})) & \vdash & \mathsf{wpi}\ (\texttt{cmpXchg}\ l\ v_3\ v_2)\ \{\Phi\} & \textsc{wp-i-cmpXchg-fail}
\end{array}
$$

$\textsc{wp-i-iter}$

$$
\mathsf{wpi}\ (f\ x) \left\{ w.\ \begin{bmatrix} \triangleright \mathsf{wpi}\ (\texttt{iter}\ f\ x')\ \{\Phi\} & \text{if } w = \texttt{inl}\ x' \\ \Phi\ y & \text{if } w = \texttt{inr}\ y \end{bmatrix} \right\} \vdash \mathsf{wpi}\ (\texttt{iter}\ f\ x)\ \{\Phi\}
$$

Figure 4.1: Interaction trees weakest preconditions

There is a new element to the familiar wp-i-get, wp-i-put, wp-i-alloc and wp-i-free, the heap requirements have a later in front of them. This reflects the fact that all of these perform a computational step, since they are defined using the `Vis` constructor. The rules as they were in chapter 2 are derivable from the new versions.

We now explain the three new rules:

wp-i-fork The fork rule requires two things. It requires a complete weakest precondition derivation for the expression we are forking, with the trivial postcondition True. This post-condition makes sense because the forked-off thread has no way to return a value in the end anyways. Besides having to take care of the forked-off thread, we also need to prove the postcondition for the current thread and finish its own derivation.

wp-i-cmpXchg-suc The wp-i-cmpXchg-suc rule is the case where the value (v_1) we expect to be on the heap, is actually there and the exchange happens. We need to prove however that v_1 is truly at location l, this is the reason we have the $\triangleright l \mapsto v_1$ hypothesis. The later modality is there because `cmpXchg` performs a computational step. We obtain new information to prove the postcondition, namely that location l now points to v_2, because the exchange happened. Finally the post condition should hold for the return value that v_1 was indeed on the heap, and True because an exchange happened.

wp-i-cmpXchg-fail The wp-i-cmpXchg-fail rule is a counterpart to the previous rule, where no exchange happens. We need to still own location l, but also have proof that the value there is different from the value we are expecting to be there. This is done by the first two hypotheses. In our postcondition we get back ownership of location l but with the value unchanged, because the exchange did not occur. The postcondition needs to hold for the value that was actually there (v_1) and False because no exchange occurred.

We finish this chapter by showing our previous programs are now expressible as itree programs. The same specs for these programs are also verifiable by similar proofs as in section 2.5 and section 3.4, since we have the exact same proof rules for `itrees`.

```
Definition prog_swap  (l k: nat): expr nat unit :=
  x ← itree.get l ;
  y ← itree.get k ;
  itree.put l y ;; itree.put k x.

Definition loop_body {A}: unit → expr nat A :=
  itree.iter (λ x, mret $ inl ()).

Definition loop_prog {A}: expr nat A :=
  loop_body ().
```

We leave the more interesting interaction trees programs for chapter 6.

4.3 Interaction trees interpreter

We need an evaluator for our expressions before we can state our adequacy theorem. An evaluator for our expressions needs:

- Access to mutable state.
- A thread-pool to store forked threads.

- A scheduler to make the interleaving of threads non-trivial.
- Fuel to ensure termination.

Therefore our top level definition is:

```
(*
  cmp: The EqDecision type class is there to be able to
       compare values on the heap for CasE.
  n  : The fuel to ensure termination.
  s  : The co-inductive scheduler to allow for interleaving of threads.
  e  : The program to run.
*)
Definition run_program {V R} {cmp: EqDecision V} (n: nat)
   (s: scheduler V R) (e: expr V R): error R.
```

The implementation (which can be found online [2]) of our interpreter translates all the commands in our program e into operations of a state monad with a thread-pool. We then ask the scheduler s what thread should take a step. We then retrieve this thread t from our thread-pool allow it to process one atomic operation then store the resulting expression back in our thread-pool. At the end we check if our main thread has terminated yet, if so the entire program stops. The entire implementation can be found in the formalization. The two interesting aspects of the interpreter for adequacy are the scheduler and the `error` type. The scheduler is defined as:

```
CoInductive scheduler V R := Scheduler {
 schedule: list (thread V R) * heap V → nat * scheduler V R
}.
```

A way to think about this is as a `stream nat` where each value depends on the state of the thread-pool and heap at that point of execution of the program. Lastly the error type is defined as:

```
Inductive error (A: Type): Type :=
| Here (a: A)
| ProgErr
| EvalErr.
```

The `error` type is similar to `option` type, but has two distinct modes of failure. For the adequacy theorem we need to know if the program failed because we ran out of fuel, or because an operation in the program itself failed. We want to rule out the program errors and allow the errors that occurred in evaluation. We will now state our adequacy statement in the next section.

4.4 Interaction trees adequacy

With the top level definition for our interaction trees interpreter established we can now give an adequacy theorem for interaction trees.

Theorem 3 *Let Ψ be a first-order predicate (rather than a separation logic assertion). If* True $\vdash$ $\mathsf{wpi}_{\mathcal{E}}\ e\ \{\Psi\}$ *and let r be the result of* `run_program` $n\ s\ e$ *then $\Psi\ x$ holds if r is* `Here` x*, or* False *holds if r is* `ProgErr`*, or finally* True *should hold if r is* `EvalErr`

[2]https://github.com/LiamClark/weakest-pre/blob/cmp-swp/theories/program_logic/evaluation.v

This gives us partial-correctness in regards to running out of fuel. The theorem gives us memory safety, because we need to prove False if the execution results in a `EvalErr`. That means a valid weakest precondition derivation rules out all memory issues. Now that we have seen interaction trees in the single-threaded setting, we move to extending our logic for the concurrent setting.

Chapter 5

Pothos model

In this chapter we provide the semantic basis for all previously seen weakest precondition inference systems. This is done by giving a definition for the weakest precondition connective in terms of the Iris separation logic. All the previously seen inference rules can be derived from these definitions for the weakest precondition connective, meaning they are no longer axiomatic. Simultaneously the adequacy theorems interface with this definition rather than having to work with the whole set of inference rules. There are established modality patterns in Iris (Jung, Krebbers, et al. 2018, Sections 6.3 and 7.3) that we can use to accommodate different language features in our expression language. This includes the later modality and the new update modalities we will introduce in this chapter. None of these patterns are new, but our application of these patterns to shallowly embedded (coinductive) monadic languages is novel. Especially the interplay between the later modality and the co-inductive types.

We also lay some groundwork for the verification of multi-threaded programs by introducing invariants and the fancy update modality This chapter covers all this in the following order:

- The basic update modality, model of the heap and the weakest precondition definition for the state monad in section 5.1.
- The Iris fixpoint operator and the weakest precondition definition for the delay monad in section 5.2.
- Invariants, the fancy update modality and the weakest precondition definition for interaction trees in section 5.3.

At the end of this chapter we should have increased confidence our inference systems are sound and that our adequacy theorems give us memory safety for our programs.

5.1 State weakest precondition model

The language feature we need to accommodate for our weakest precondition definition for the state monad is mutable state. Iris is parametric in regard to separation logic resources, which means we have to model our concept of a heap and the ($\mapsto$) points to connective. We do this in the standard Iris way (Jung, Krebbers, et al. 2018, Section 6.3.2). We only give the top level definitions here, but the full definitions can be found in the Coq implementation. Up until this point we have assumed that Iris knows about our heap and can reason about it from the get go, this is not actually the case. Rather than making assumptions about how the heap works, Iris allows the user to make Iris aware of what their definition of the heap is. Let us recall that the heap of state monad was defined as: σ : `gmap nat A`. We interpret this map σ into an iProp by means of the state-interpretation predicate: $SI(\sigma)$: `gmap nat A`

→ iProp. We will leave the *SI* predicate opaque and see it as representing ownership of the heap. The definition of *SI* also allows for the definition of a points-to connective. To perform updates to the heap we require the basic update modality, which allows updates to all sorts of resources in Iris. We now proceed by first introducing this modality, presenting the definition of weakest precondition connective for state in subsection 5.1.2 and finally explain our top level usage of the *SI* predicate.

5.1.1 Basic update modality

The basic update modality allows us to perform updates to the Iris version of the heap. Jung, Krebbers, et al. (2018) gives the following intuition for the basic update modality: The update modality $\dot{\Rrightarrow} P$ provides a way to talk about the resources we could own after we update what we do own. When performing a proof this means that: as long as our conclusion is guarded by a $\dot{\Rrightarrow}$ we can remove a guard from our hypotheses. When we are done removing guards, we are always free to remove the guard from our conclusion. The basic update modality has the following rules:

$$\overset{\text{UPD-INTRO}}{P \vdash \dot{\Rrightarrow} P} \qquad \overset{\text{UPD-TRANS}}{\dot{\Rrightarrow} P \vdash \dot{\Rrightarrow}\dot{\Rrightarrow} P} \qquad \overset{\text{UPD-MONO}}{\dfrac{P \vdash Q}{\dot{\Rrightarrow} P \vdash \dot{\Rrightarrow} Q}} \qquad \overset{\text{UPD-FRAME}}{Q * \dot{\Rrightarrow} P \vdash \dot{\Rrightarrow}(Q * P)}$$

- The UPD-INTRO rule allows us to introduce the basic update modality. In essence this means we say we do not need to do any further updates and we can remove the guard from our conclusion.
- The UPD-TRANS rule allows us to join two basic update modalities into one. Performing two updates after each other is the same as performing one bigger update.
- The UPD-MONO rule allows us to remove the basic update modality guard from one of our hypothesis, much like the ▷-MONO rule.
- The UPD-FRAME rule gives us the ability to include more resources to our update.

As an example of these rules we prove a derived rule that might fit intuition better. The rule is:

$$\overset{\text{UPD-DERIV}}{\dfrac{P * Q \vdash \dot{\Rrightarrow} R}{(\dot{\Rrightarrow} P) * Q \vdash \dot{\Rrightarrow} R}}$$

The UPD-DERIV rule strips a basic update modality from a hypothesis if we have a basic update modality guarding our conclusion. The guard on the conclusion is not removed, meaning we can keep applying the rule to remove guards from all hypotheses.

We now give the derivation of the UPD-DERIV rule:

$$\dfrac{\dfrac{\dfrac{}{(\dot{\Rrightarrow} P) * Q \vdash \dot{\Rrightarrow}(P * Q)}\ \begin{matrix}\text{SEP-COMM}\\ \text{UPD-FRAME}\end{matrix} \qquad \dfrac{P * Q \vdash \dot{\Rrightarrow} R}{\dot{\Rrightarrow}(P * Q) \vdash \dot{\Rrightarrow}\dot{\Rrightarrow} R}\ \text{UPD-MONO}}{(\dot{\Rrightarrow} P) * Q \vdash \dot{\Rrightarrow}\dot{\Rrightarrow} R}\ \vdash\text{-TRANS} \qquad \dfrac{}{\dot{\Rrightarrow}\dot{\Rrightarrow} R \vdash \dot{\Rrightarrow} R}\ \text{UPD-TRANS}}{(\dot{\Rrightarrow} P) * Q \vdash \dot{\Rrightarrow} R}\ \vdash\text{-TRANS}$$

5.1.2 State weakest precondition model

We now present the definition of weakest precondition connective for state:

$$\begin{aligned}\mathsf{wps}\ e\ \{\Phi\} \triangleq\ & \forall \sigma, SI(\sigma) \mathrel{-\!\!*} \\ & \dot{\Rrightarrow} \exists a\ \sigma',\ \texttt{runState e}\ \sigma = \texttt{Some (a,}\ \sigma'\texttt{)} \\ & \qquad * SI(\sigma') \\ & \qquad * \Phi\ a\end{aligned}$$

The weakest precondition for state uses a big-step style semantics, meaning we run the entire computation and reason about the result it produces. Before we can run the program the state monad requires a heap (of type `gmap nat A`), this is present in our definition as σ. We tie σ to Iris with the before mentioned *SI* predicate: For now this reads as: we have exclusive ownership of the entire heap σ.

Now we can run the computation using σ, we then require the computation to succeed with a return value a and a final heap σ'. Because we require the computation to succeed this definition gives us memory safety (any invalid memory access would return a `None`). The second clause keeps track of the fact that we now own a new heap, allowing us to keep track of the heap as it progresses though our computation. The third clause requires the program to terminate with a value for with the postcondition holds giving us correctness.

The last ingredient in the definition is the basic update modality just before the existential-quantifier. The standard Iris way to enable updates to the heap is to place it between receiving ownership of the old heap and having to provide ownership of the new heap. This pattern makes the weakest precondition rules for get, put, alloc and free derivable. We can see the reason for this in the following Lemmas regarding the *SI* predicate (we assume them here, but they are all derivable within Iris). The derivation for the weakest precondition version of these simply lift these Lemmas to the weakest precondition level.

$$SI\ \sigma \mathrel{-\!\!*} l \mapsto v \mathrel{-\!\!*} (\sigma\ \texttt{!!}\ \texttt{n} = \texttt{Some v}) \qquad \textsc{si-get}$$

$$SI\ \sigma \mathrel{-\!\!*} l \mapsto v \mathrel{-\!\!*} \dot{\Rrightarrow} SI\ (\texttt{<[}l\ \texttt{:=}\ w\texttt{]>}\sigma) * l \mapsto w \qquad \textsc{si-put}$$

$$SI\ \sigma \mathrel{-\!\!*} l \mapsto v \mathrel{-\!\!*} \dot{\Rrightarrow} SI\ (\mathsf{delete}(\sigma, l)) \qquad \textsc{si-free}$$

$$SI\ \sigma \mathrel{-\!\!*} \dot{\Rrightarrow} SI\ (\texttt{<[}k\ \texttt{:=}\ v\texttt{]>}\sigma) * l \mapsto v \qquad \textsc{si-alloc}$$

where k is the first unused key in σ

- The SI-GET lemma is the only one without a basic update modality, because retrieving an element from the heap does not alter the heap. It does tie together the points-to connective with our heap. If we own a location we also know it is present in our heap.
- The SI-PUT lemma allows us to write to a location we own, reflecting this change in the heap and in the new points-to connective obtained. Note that this lemma and all further lemmas are guarded by a basic update modality.
- The SI-FREE lemma is similar to SI-PUT, but rather than updating the heap it removes from the heap. note that it does not giving ownership of the location back.
- Finally the SI-ALLOC lemma gives us ownership of a new location and updates the heap accordingly.

We can now also see why the $\dot{\Rrightarrow}$ is required in the weakest precondition definition for state. We need it to perform updates to our model of the heap, which we require for the derivation of our weakest precondition inference rules.

5.2 Delay weakest precondition model

Next is the definition of weakest precondition for the delay monad. The language feature we need to accommodate is non-termination, this means our definition for weakest precondition has a case distinction whether the program is finished or not. To handle the non-termination we change the formulation of our weakest precondition to use a small-step semantics style, compared to the big-step style seen in subsection 5.1.2. The standard Iris pattern here is to recursively apply the weakest precondition guarded by the later modality. Since the delay monad does not incorporate a heap, the basic update modality and other machinery seen in section subsection 5.1.2 is absent here. We now give the definition:

$$\mathsf{wpd}\ e\,\{\Phi\} \triangleq \left\{ \begin{array}{ll} \Phi\ x & \text{if } e = \texttt{Answer}\ x \\ \triangleright \mathsf{wpd}\ e'\,\{\Phi\} & \text{if } e = \texttt{Think}\ e' \end{array} \right\}$$

The first thing to see is the before mentioned case distinction in the definition. If we see an `Answer` constructor we know our program has terminated and we simply require the post-condition to hold for the value returned. The more interesting case is if we see a `Think` constructor, because now our program might not terminate. Iris can deal with this by means of a guarded fixpoint operator (Jung, Krebbers, et al. 2018, Section 5.6), this operator allows recursive occurrences as long as they are guarded by the $\triangleright$ modality. The recursive occurrence of: $\triangleright \mathsf{wpd}\ e\,\{\Phi\}$ is actually handled by the fixpoint operator. The usage of the $\triangleright$ modality to guard the recursive occurrence corresponds with our notion of computational step from section 3.3, namely that every time we encounter a `Think` constructor this takes us to the next step of computation. With this definition we can require the postcondition to hold for terminating programs, encode our notion of computational steps and gracefully allow for partial-correctness of non-terminating programs.

5.3 Itree weakest precondition model

We now turn to the last weakest precondition definition, the one for interaction trees. The language features we need to accommodate are mutable state, non-termination and concurrency. For the mutable state and non-termination we combine the techniques from the previous sections. We have more work to do for concurrency, namely our weakest precondition inference rules do not support concurrency (yet). We require knowledge of more Iris constructs before we can add inference rules for concurrency. The flaw with our weakest precondition inference rules is that they do not support concurrent programs that communicate. The way to support sharing across threads in Iris is by means of invariants. To work with invariants in Iris we need to work with the fancy update modality rather than the basic update modality we saw in subsection 5.1.2. For the rules we have seen so far we could give a definition for the weakest precondition for interaction trees using just the basic update modality, but in chapter 6 we want to use invariants for our verification. To make this all align: we first explain invariants and the fancy update modality, give a definition for weakest precondition using the fancy update modality and finally expand the inference rules for interaction trees to make use of the new machinery.

5.3.1 Invariants

An invariant is a property that holds at all times. Each thread may assume the invariant holds before it executes, also we must ensure the invariants still hold after execution of a computational step. In Iris this means we can register an `iProp` as an invariant, Iris then keeps track of all established invariants and enforces that they always hold. To keep track of all the invariants, they are registered in a namespace. We now give the connective for establishing a proposition P with name $\mathcal{N}$ as an invariant: $\boxed{P}^{\mathcal{N}}$. The rules for invariants are present in Figure 5.1. The INV-DUP rule is what allows the invariant to be shared across threads, namely we can duplicate the invariant. Whenever a new thread spawns, we can duplicate the invariant and both threads can obtain their own copy.

The main rule we will use in verifications is the following WP-I-INV:

$$\frac{\overset{\text{WP-I-INV}}{\triangleright P \vdash \mathsf{wpi}_{\mathcal{E}\setminus\mathcal{N}}\ e\ \{w.\, \triangleright P * \Phi\ w\}} \qquad \mathit{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{wpi}_{\mathcal{E}}\ e\ \{\Phi\}}$$

The rule allows to access an invariant around one atomic expression as long as we give back the invariant afterwards. The subscript on the weakest precondition connective is called a mask and will be covered in subsection 5.3.2. The WP-I-INV rule can be derived from INV-ACCESS and WP-ATOMIC.

5.3.2 Fancy update modality

The final piece of machinery we introduce for working with invariants is the fancy update modality. We use the fancy update modality as a basic update modality, but with the ability to access invariants. The fancy update modality is written as: ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$. The superscripts are called masks, they are sets of names of invariants that are currently available. Thus we can read a ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$ modality as: we could have resources P after performing an update that changes the set of available invariants from $\mathcal{E}_1$ to $\mathcal{E}_2$. The masks on the fancy update modality do not actually contain the invariant they are simply tokens to ensure the proper use of invariants (for example preventing opening the same invariant twice).

The first four rules in Figure 5.1 are updated versions of the rules for the basic update modality. The FUPD-INTRO rule gives us a way to introduce the fancy update modality if the incoming mask is the same as the outgoing mask. In similar fashion the FUPD-TRANS rule allows composition of fancy update modalities if the outgoing mask of the first, matches the incoming mask of the second. The FUPD-MONO has nothing new going on, and then finally the FUPD-FRAME does two things. It allows other iProps to enter under then fancy update modality and allows widening of the mask through $\mathcal{E}_f$. The FUPD-BUPD is there to degrade a fancy update modality into a normal update in case where the is not required. Finally we have three new rules.

Inv-alloc and inv-dup If we want to allocate an iProp P as an invariant, then P should hold. Then INV-DUP states that if we have P as an invariant we can obtain copies of it.

Inv-access The INV-ACCESS rule is what all the newly introduced machinery is for. It reads as follows: If we have an invariant $\boxed{P}^{\mathcal{N}}$ and the name $\mathcal{N}$ is still available in our mask then, we can perform a fancy update that removes the name $\mathcal{N}$ from our mask. After the fancy update we gain access to the payload of the invariant: P at the next step of computation. We also gain a second iProp: ($\triangleright P \twoheadrightarrow {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}}\ \mathsf{True}$) this states that if we give up the invariant payload we obtain a fancy update modality that we can eliminate that re-establishes the name $\mathcal{N}$ in the mask.

$$\textsc{fupd-intro}\quad P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\, P$$

$$\textsc{fupd-trans}\quad {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3}\, P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3}\, P$$

$$\textsc{fupd-mono}\quad \frac{P \vdash Q}{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, Q}$$

$$\textsc{fupd-frame}\quad Q * {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}{\Rrightarrow}^{\mathcal{E}_2 \uplus \mathcal{E}_f}\, (Q * P)$$

$$\textsc{fupd-bupd}\quad \dot{\Rrightarrow} P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\, P$$

$$\textsc{inv-alloc}\quad \triangleright P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\, \boxed{P}^{\mathcal{N}}$$

$$\textsc{inv-access}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}}\, (\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}}\, \mathsf{True}))}$$

$$\textsc{inv-dup}\quad \boxed{P}^{\mathcal{N}} \vdash \boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}$$

$$\textsc{wp-atomic}\quad \frac{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, \mathsf{wpi}_{\mathcal{E}_2}\, e \left\{ {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_1}\, \Phi \right\} \qquad \mathit{atomic}(e)}{\mathsf{wpi}_{\mathcal{E}_1}\, e\, \{\Phi\}}$$

Figure 5.1: Rules for the fancy update modality

That leaves one question, how do we eliminate a fancy update modality? We give a derived rule for this with its derivation:

$$\textsc{fupd-change-mask}\quad \frac{R * P \vdash {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3}\, Q}{R * {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3}\, Q}$$

$$\dfrac{\dfrac{\dfrac{R * P \vdash {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3}\, Q}{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, (R * P) \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3}\, Q}\ \textsc{fupd-mono}}{R * {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3}\, Q}\ \vdash\textsc{-trans fupd-frame}}{R * {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3}\, Q}\ \vdash\textsc{-trans fupd-trans}$$

The final insight required is that if the rest of our proof is guarded by a fancy update modality with different masks, we can not progress past this fancy update modality with the fupd-intro rule, until the two mask match. Thus what the second half of the inv-access rule really achieves is a way to progress to the next part of the proof, by re-establishing the right masks if we give up the invariant. Ensuring that the invariant is only opened for one atomic operation.

wp-atomic The masks around wp-atomic allows us to access an invariant around the expression e, as long as we restore the mask properly in the post condition. Meaning the invariant can not stay open for longer than the single expression.

5.3.3 Itree weakest precondition definition

The weakest precondition definition for interaction trees is given in two parts: the definition itself and a helper predicate to constrain the heap for each of our operations from our environment. The definition is as follows:

$$\mathsf{wpi}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq \left\{ \begin{array}{ll} {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\, \Phi\ x & \text{if } e = \texttt{Answer}\ x \\ {}^{\mathcal{E}}{\Rrightarrow}^{\varnothing}\, \triangleright {}^{\varnothing}{\Rrightarrow}^{\mathcal{E}}\, \mathsf{wpi}_{\mathcal{E}}\, e'\, \{\Phi\} & \text{if } e = \texttt{Think}\ e' \\ {}^{\mathcal{E}}{\Rrightarrow}^{\varnothing}\, \triangleright {}^{\varnothing}{\Rrightarrow}^{\mathcal{E}}\, (\mathsf{wpi}_{\mathcal{E}}\, e'\, \{\Phi\} * \mathsf{wpi}_{\top}\, e_f\, \{\_.\mathsf{True}\}) & \text{if } e = \texttt{Fork}\ e_f\ e' \\ \forall \sigma, \mathit{SI}(\sigma) \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\varnothing}\, \triangleright {}^{\varnothing}{\Rrightarrow}^{\mathcal{E}}\, \exists \sigma'\ v, & \\ \qquad \mathit{CP}(c, \sigma, \sigma', v) * \mathit{SI}(\sigma') * \mathsf{wpi}_{\mathcal{E}}\, (k\ v)\, \{\Phi\} & \text{if } e = \texttt{Vis}\ c\ k \end{array} \right\}$$

The common thing across all these definitions is the standard Iris pattern of modalities: ${}^{\mathcal{E}}{\models\!\!\!\Rrightarrow}^{\emptyset} \triangleright {}^{\emptyset}{\models\!\!\!\Rrightarrow}^{\mathcal{E}}$. The fancy update modalities are there to allow reasoning disregarding invariants for one operation, as long as we restore them at the end. The later modality in the middle is there to state that these operations take a step of computation as seen in section 5.2. Finally there is also added value of having update modalities on both sides of the later modality. The two modalities do not commute and just having one makes it impossible to derive all the weakest precondition rules that affect the heap.

Answer and Think The first two cases should be straightforward at this point. If we encounter an `Answer` constructor, then the postcondition should hold for the value inside the `Answer` constructor. This is exactly as it was in `wpd`. The case for the `Think` constructor is similar to the one we saw in `wpd`, however the modalities guarding the recursion have been expanded. First we only had a later modality, now we have the entire pattern of fancy update modalities and a later modality.

Fork The `Fork` constructor starts off with the modality pattern and then does a binary recursion. Requiring a weakest precondition for our current thread (this is the one for which our postcondition: Φ should hold) and a weakest precondition for the forked-off thread. The weakest precondition for the forked-off thread has the postcondition `True` because it does not return a value, furthermore it has a mask with all invariants labelled as available. The choice of the $\top$ mask works, because the soonest the forked-off thread can start running is at the next step of computation.

Vis The case for the `Vis` constructor has to deal with operations that affect the heap. Therefore we re-introduce the machinery seen in `wps`. We take in an $SI(\sigma)$ and have SI hold for an updated heap σ'. We also require there to be a weakest precondition for the continuation k. The final ingredient is the command predicate (command predicate). The command predicate constraints the value returned for every command in our environment, at the same time it also puts constraints on the heap for the command to execute successfully. The definition of the predicate will be covered next.

5.3.4 Command predicate

The command predicate is a ”normal” predicate rather than an `iProp`. The command predicate is defined in Figure 5.2. The predicate has four parameters: The command, the heap before execution of the command (σ), the heap after execution of the command (σ') and the value fed into the continuation (v). Since the command predicate is not an `iProp` it works directly with the maps that represent our heap rather than the interpretation of them into Iris (SI).

GetE For the `GetE` case we simply require the direct lookup of l in σ to succeed with the return value v. We also record the fact that the `GetE` operation does not change the heap by requiring σ and σ' to be equal.

PutE In the `PutE` case we require the location we are writing to to already be present in σ. We then relate σ' to σ by performing the exact same put operation directly on the map itself.

AllocE In the `AllocE` case we have a location l that we will eventually yield to the continuation. We require l to be fresh in σ, more specifically it should be the lowest unused key in σ. We then perform this allocation on σ in similar fashion as `PutE`.

$$\frac{\sigma\,!!\,l = \texttt{Some}\ v \qquad \sigma' = \sigma}{CP\ (\texttt{GetE}\ l)\ \sigma\ \sigma'\ v} \qquad \frac{l \in \sigma \qquad \sigma' = \texttt{<[l := v']>}\sigma}{CP\ (\texttt{PutE}\ l\ v')\ \sigma\ \sigma'\ \_}$$

$$\frac{\mathsf{fresh}(\sigma, l) \qquad \sigma' = \texttt{<[l := v]>}\sigma}{CP\ (\texttt{AllocE}\ v)\ \sigma\ \sigma'\ l} \qquad \frac{l \in \sigma \qquad \sigma' = \mathsf{delete}(\sigma, l)}{CP\ (\texttt{FreeE}\ l)\ \sigma\ \sigma'\ \_}$$

$$\frac{\sigma\,!!\,l = \texttt{Some}\ v1 \qquad v_{ret} = v1 \qquad \sigma' = \texttt{<[l := v2]>}\sigma}{CP\ (\texttt{CpmXchgE}\ l\ v1\ v2)\ \sigma\ \sigma'\ v_{ret}\ \mathsf{true}}$$

$$\frac{\sigma\,!!\,l = \texttt{Some}\ x \qquad v_{ret} = x \qquad \sigma = \sigma' \qquad x \neq v1}{CP\ (\texttt{CpmXchgE}\ l\ v1\ v2)\ \sigma\ \sigma'\ v_{ret}\ \mathsf{false}}$$

Figure 5.2: The inductively defined command predicate.

FreeE The `FreeE` case is quite straightforward, the location are freeing must be present in the old heap. We require the new heap to be equal to σ with a delete operation performed on it.

CmpXchgE There are two cases present for the `CmpXchgE` command, depending on whether the operation succeeds or fails. This can be seen as the last argument to command predicate with true indicating success and false indicating failure. To re-iterate $v1$ is the value we expect to be at location l already and $v2$ is the value we want to put at location l. Finally we also always return the value that was on the heap at location l before the operation as v_{ret}.

- In the success case the lookup of l in σ must be equal to $v1$ (otherwise the operation would not be a success). Therefore we can constrain the return value v_{ret} to be equal to $v1$. Since this is the successful case where the update to the heap is performed we update σ in the same way as for `PutE`.
- In case of failure the lookup of l in σ should still succeed, but with a value that is not equal to $v1$. This also means we cannot perform a `CmpXchgE` operation on an unallocated location. We constrain v_{ret} to be equal to the value that was at location l. Finally the heap should remain unchanged therefore σ' should be equal to σ.

Chapter 6

Case study: Concurrency

We now apply all that we have established in our previous chapters to our final example, the verification of a multi-threaded program. The program will be an interaction trees program, using invariants and spin-locks. We will introduce one last Iris concept called ghost-state to reason about the resources within the invariant. This case study serves three purposes:

- To show the expressiveness of our interaction trees based expression language.
- To show our program logic is non-trivial.
- That the standard Iris patterns for concurrent programs work for our program logic.

The program we will be verifying is a bank account that multiple threads will withdraw from. The withdrawing is therefore guarded by a spin-lock to make sure no thread-races occur. Finally in the case of overdraft the program will crash, therefore our verification will also prove that no overdraft ever happens.

In this chapter we first give the source code of the program we want to verify (section 6.1). Then we give the implementation and specifications of our spin-lock library (section 6.2). After the spin-lock library we introduce our new Iris concept: ghost-state (section 6.3). We give specs for the helper functions of our programs and outline their proofs (section 6.4). Finally we give a spec for the whole program and give an outline of its proof (section 6.5).

6.1 The program

Before we can write our program we need to figure out the type of values we want on our heap. In this example we need natural numbers to represent the bank balance. Besides that we need two states for our lock: `Locked` and `Unlocked`. Our heap cell definition therefore is:

```
Variant cell :=
|Locked
|UnLocked
|Value (n: nat).
```

We now give the top level program:

```
Definition as_value (c: cell): expr cell nat :=
  match c with
  | Locked ⇒ itree.fail
  | UnLocked ⇒ itree.fail
  | Value n ⇒ mret n
  end.

Definition withdraw (amount: nat) (balanceLoc: loc): expr cell bool :=
  balanceCell ← get balanceLoc ;
  balance ← as_value balanceCell ;
  if (amount ≤? balance)
  then put balanceLoc (Value (balance - amount)) ;; mret true
  else mret false.

Definition withdraw_locked (amount: nat) (lockLoc: loc)
  (balanceLoc: loc): expr cell () :=
    acquire lockLoc ;;
    b ← withdraw amount balanceLoc ;
    if b : bool then release lockLoc else itree.fail.

Definition bank_prog: expr cell () :=
  balanceLoc  ← alloc (Value 100) ;
  lockLoc     ← new_lock ;
  fork (
    withdraw_locked 5 lockLoc balanceLoc ;;
    withdraw_locked 25 lockLoc balanceLoc
  ) ;;
  withdraw_locked 20 lockLoc balanceLoc ;;
  withdraw_locked 2  lockLoc balanceLoc ;;
  withdraw_locked 10 lockLoc balanceLoc ;;
  mret tt.
```

The code has three layers, first we have `withdraw` which removes an amount from a cell at a location `balanceLoc` if the balance is sufficient. On top of `withdraw` we have `withdraw_locked` which combines the `withdraw` operation with a lock from our spin-lock library, making the operation thread-safe. Note that if the balance is insufficient we exit the entire program by calling fail. Finally our third layer `bank_prog` allocates an initial balance and lock. It forks-off a thread and both threads withdraw form the balance through the lock multiple times.

6.2 Spin-lock library

We can now give definitions for our spin-lock. The public api will have three top level constructs: `new_lock`, `acquire` and `release`.

```
Definition new_lock: expr cell loc :=
  alloc UnLocked.

Definition try_acquire (l: loc): expr cell bool :=
  snd <$> cmpXchg l UnLocked Locked.

Definition acquire (l: loc): expr cell () :=
  itree.iter
    (λ _, try_acquire l ≫= λ (b : bool), if b then mret $ inr $ ()
                                          else mret $ inl $ ()
    )
    tt.
```

```
Definition release (l: loc): expr cell () :=
  put l UnLocked.
```

The `try_acquire` function is the heart of the spin-lock, it uses the `CmpXChg` operation on the location of our lock and expects an `Unlocked` to be present at that heap location. If that is correct, we actually acquire the lock and write `Locked` to the heap and returing a boolean whether or not our exchange suceeded. The writing of `Locked` prevents the `try_acquire` operation from any other thread from succeeding.

The spin part of our spin-lock is in `acquire` which applies our iteration-combintaor to the `try_acquire` operation. Meaning the thread will hang until we acquire the lock. Finally we have the release operation, which simply overwrites the lock with the `Unlocked` value.

Now that we have established the definitions for spin-locks, we give specifications for them:

$$R \mathrel{-\!\!*} (\forall l, \mathit{is\_lock}(l, R) \mathrel{-\!\!*} \Phi\ l) \mathrel{-\!\!*} \mathsf{wpi}_{\top}\ \texttt{new\_lock}\ \{\Phi\}$$

$$\mathit{is\_lock}(l, R) \mathrel{-\!\!*} (R \mathrel{-\!\!*} \Phi\ ()) \mathrel{-\!\!*} \mathsf{wpi}_{\top}\ \texttt{acquire}\ \ l\ \{\Phi\}$$

$$\mathit{is\_lock}(l, R) \mathrel{-\!\!*} R \mathrel{-\!\!*} \Phi\ () \mathrel{-\!\!*} \mathsf{wpi}_{\top}\ \texttt{release}\ \ l\ \{\Phi\}$$

These specifications read as follows:

- To allocate a lock we need to give it the payload R we want to store it in. The `new_lock` operation will then initialize a lock at a new location l. In the postcondition we recieve the newly allocated location and the fact that there is a lock there with R as the payload.
- To acquire a lock we first need to show there is a lock at location l , when we acquire it we gain access to the payload R, that was stored inside the lock, in the postcondition.
- Finally to release a lock we need to show that there is a lock at location l and give up the payload to store it back into the lock.

We can now give the definition for *is_lock*:

$$\mathit{lock\_inv}(l, R) \triangleq \exists (c : \texttt{cell}), l \mapsto c * \begin{bmatrix} \mathsf{True} & \text{if } c = \texttt{Locked} \\ R & \text{if } c = \texttt{UnLocked} \\ \mathsf{False} & \text{if } c = \texttt{Value \_} \end{bmatrix}$$

$$\mathit{is\_lock}(l, R) \triangleq \boxed{\mathit{lock\_inv}(l, R)}^{\mathcal{L}}$$

First we have *lock_inv* which requires ownership of location l to some cell c. Then there is a case distinction for each possible value that c could be. If the cell is `UnLocked` we store the payload R. If the cell is `Locked` we store nothing (True) since we have given up the payload to the thread which acquired the lock. Finally we want to rule out that the cell c can be anything else thus for the case `Value n` we store False to rule it out.

On top of that *is_lock* wraps *lock_inv* by registering it in Iris as an invariant, note that this makes the knowledge that there is a lock sharable across threads. But the payload will only ever be accessed by one thread at the same time, by acquiring and releasing the lock. The $\mathcal{L}$ is a namespace dedicated to the invariants of locks.

6.3 Ghost state

We now introduce the ghost state required for our verification, we give a top level abstraction for this and do not cover any of the internals. The construct we will use here is the authoritative camera on natural numbers (Jung, Krebbers, et al. 2018, Section 6.3.2). The authoritative camera has two constructs, the first expresses knowledge about the whole (written $\bullet$). Then there is the idea of fractional ownership, meaning a part of the whole (this is written as $\circ$). We will use the $\bullet$ to represent ownership of the entire initial bank balance. We will split this into fragments $\circ$ for each withdrawal made.

This leaves us with the following rules for our ghost state:

AUTH-FRAG-LT
$$\boxed{\bullet m}^{\gamma} \mathrel{-\!\!*} \boxed{\circ n}^{\gamma} \mathrel{-\!\!*} m \leqslant n$$

FRAG-SPLIT
$$\boxed{\circ(n+m)}^{\gamma} \mathrel{-\!\!*} \boxed{\circ n}^{\gamma} * \boxed{\circ m}^{\gamma}$$

AUTH-FRAG-DEDUCT
$$\boxed{\bullet n}^{\gamma} \mathrel{-\!\!*} \boxed{\circ m}^{\gamma} \mathrel{-\!\!*} \boxed{\bullet(n-m)}^{\gamma}$$

AUTH-FRAG-ALLOC
$$\forall n, \dot{\Rrightarrow} \exists \gamma, \boxed{\bullet n * \circ n}^{\gamma}$$

These rules in order state the following:

- If we have ownership of the whole and ownership of a fragment, the value in the fragment is always less than equal than the full thing. We will use this rule to proof that the withdrawal we are making can never overdraft the account.
- If we have a fragment of a sum of two numbers we can always split it into two fragments one for each part of the sum.
- If we give up ownership of a fraction, we can update our knowledge of the whole to decrease by the size of the fraction.
- Finally we can allocate ghost state for any number n and get the full ownership for n and fractional ownership of n.

6.4 Withdraw specifications

Now we can turn our attention to the actual program we want to verify. Let us look at the specs for `withdraw` and `withdraw_locked`

$$m \leqslant n \mathrel{-\!\!*} l \mapsto \texttt{Value}\ n \mathrel{-\!\!*} (l \mapsto (\texttt{Value}\ (n-m)) \mathrel{-\!\!*} \Phi\ \texttt{true}) \mathrel{-\!\!*} \mathsf{wpi}_{\top}\ \texttt{withdraw}\ m\ \{\Phi\}$$

$$\mathit{lock\_payload}(l) \triangleq \exists n. \boxed{\bullet n}^{\gamma} * l \mapsto (\texttt{Value}\ n)$$

$$\mathit{is\_lock}(\mathit{llock}, \mathit{lock\_payload}(\mathit{lbal})) \mathrel{-\!\!*} \boxed{\circ m}^{\gamma} \mathrel{-\!\!*} \Phi\ () \mathrel{-\!\!*} \mathsf{wpi}_{\top}\ \texttt{withdraw\_locked}\ m\ \mathit{llock}\ \mathit{lbal}\ \{\Phi\}$$

First we have the specification for `withdraw`. The specification requires proof that the amount we withdraw is less than the current balance, therefore ruling out overdrafts. The specification requires ownership of the balance location and then returns ownership with the reduced balance in the postcondition. Since we have ruled out overdrafts `withdraw` always returns true for which our postcondition should hold.

We establish *lock_payload* as the payload we will store within the lock invariant. The payload owns the location of our balance, but it does not know what our balance is. Alongside the balance it has full ownership of the ghost state representing the balance.

The *lock_payload* is then used by the specification for `withdraw_locked`, which requires a lock at location *llock* with payload *lock_payload* for location *lbal*. The final argument is fractional ownership of the amount m we are going to withdraw. We want to use the spec for `withdraw` to prove the spec for `withdraw_locked`. All of these come together in the proof in the following way: First we access the payload to gain ownership of *lbal* and the full ownership of our ghost state n. We can then combine our two pieces of ghost state with AUTH-FRAG-LT to gain proof that $m \leqslant n$. This makes it possible to apply the `withdraw` spec. After which we can update our payload using AUTH-FRAG-DEDUCT. Then we can restore the payload for the updated balance and give it back to the lock with the `release` spec.

6.5 Driver program

The last step in completing our verification is verifying the driver program. We want to give it the following specification:

$$\mathsf{True} \vdash \mathsf{wpi}_{\top}\ \texttt{bank\_prog}\ \{x.x = tt\}$$

The specification guarantees our program executes without memory issues and no overdraft occurs. The proof is structured as follows, we first allocate full and fractional ghost ownership for our entire balance of 100 by means of: AUTH-FRAG-ALLOC. We then allocate our balance and lock, using the ownership of the balance and our full ghost ownership as the payload for the lock. Leaving the fractional ownership in our context, to be split up amongst the threads with FRAG-SPLIT and every time a withdrawal needs to be made. Finally we split the verification to the two threads using WP-I-FORK and repeatedly apply the spec for `withdraw_locked` to finish the proof. The full verification can be found in the formalization. We have now seen the application of a fair share of Iris constructs to our interaction trees language.

Chapter 7

Related work

We start out by looking at the verification possibilities of the interaction trees (Xia et al. 2020) development first and compare it to the Iris based verification of Pothos. Staying with interaction trees we cover Dijkstra Monads Forever (Silver and Zdancewic 2021) as they develop a program logic for interaction trees. As mentioned in the introduction there are a lot of existing projects that allow for formal reasoning about effectful programs in a shallowly embedded language. Of these projects we will cover: SteelCore (Swamy, Rastogi, et al. 2020), Predicate transformer (Swierstra and Baanen 2019) and fine grained concurrent separation logic (Sergey, Nanevski, and Banerjee 2015). There are two more related projects we cover that are not shallow embeddings, but are related to Pothos through the separation logic aspect. The projects are: Goose & Perennial (Chajed et al. 2019) and A Separation Logic for Effect Handlers (Vilhena and Pottier 2021).

We will cover interaction trees first (section 7.1) since it will introduce information required for the comparison with other shallowly embedded projects. We cover Dijkstra monads forever next (section 7.2) because they are so closely related to interaction trees . We cover the three shallowly embedded projects projects next (section 7.3, section 7.4 and section 7.5). We then cover a separation logic for algebraic effects (section 7.6) as it relates to Iris and is an avenue for future work for Pothos. Finally we cover Perrenial (section 7.7) which relates closely to Iris but is a bit further removed from Pothos.

7.1 Interaction trees

We have used interaction trees extensively throughout this thesis, however we have only used weakest preconditions to reason about them. The interaction trees library has its own framework for reasoning about interaction trees programs. Their framework is based upon the principle of weak bisimulation and is generic in the effects used. Bisimulation is a notion of equivalency based on an external view of two programs. In the case of interaction trees, this means we only look at the commands issued to the environment. Two programs are bisimilar iff they execute the same `Vis` commands. Two programs that take a different amount of recursive steps in issuing these commands (meaning a different amount of `Think` nodes) are still considered bisimilar. There is a rich set of laws for interaction trees programs based on bisimulation. Sadly the notion of bisimulation does not give us the following property when relating it with our Iris based reasoning: $\texttt{e} \cong \texttt{e'} \rightarrow \mathsf{wpi}\ e\ \{Q\} \rightarrow \mathsf{wpi}\ e'\ \{Q\}$. If we have a weakest pre for a program e and we know another program e' is bisimilar to our original program e, then that does not guarantee there is a valid weakest pre for e'. The issue is that in our system `Think` nodes are meaningful, because they relate directly to the step index. Removing them is therefore not possible and therefore the above property is false.

Interaction trees also include a recursion combinator (called `mrec`) based on representing recursive calls as an effect. It is a rich combinator and a evolution of "Turing- Completeness

Totally Free" (McBride 2015). The original approach is to represent recursive calls as an effect in the free-er monad. That is the commands will be exactly the input of the function and the response the output, meaning we have an oracle for our function. Our function is a description of what the recursion should look like rather than a performance of this computation. The tying of the recursive knot happens when we interpret the entire function (program). A fuel based evaluator will replace the effect call with the body of our function once for every level of fuel we have. Tying the recursive knot is at evaluation time means we can only have one recursive point within our entire program.

The interaction trees `mrec` improves upon the original idea by adding mutual recursion and multiple points of recursion. This is possible because interaction trees themselves are coinductive, meaning the handler can occur within the program rather than during execution. Giving us access to multiple points of general recursion when we desire so. The signature for the recursion oracle is placed within a single effect to be handled, rather than on the entire set of commands. The issue with using `mrec` in Pothos is that we used a closed effect environment. Since the recursion is represented by an effect it naturally requires the effect environment to be extensible. Providing a definition of a weakest precondition connective for an extensible free monad is challenging and out of scope for this thesis.

7.2 Dijkstra Monads Forever

Dijkstra monads (Swamy, Weinberger, et al. 2013; Maillard et al. 2019) are a development for the verification of effectful programs (and they are deeply related to the F-star proof assistant). The idea is to have a monad model an effect (known as the base monad `M`) and have a companion monad to carry specifications (the specification monad `W`) for the computational monad `M`. The specification monads are all based on the continuation monad specialized to propositions: `(A → Prop) → Prop`. A Dijkstra monad is then the combination of these two monads and an monad morphism `h`. Values of the Dijkstra monad will then be computations `M` that uphold the specification given by the specification monad `W`, with the specification arising from the translation of the computation through monad morphism `h`. This gives Dijkstra monads the same extensibility as predicate transformers (section 7.3), however they go one step further and provide constructions to translate both the free monad and algebraic effects into their monad morphism setting. They have also applied this construction for extensible specifications to interaction trees (Silver and Zdancewic 2021). Dijkstra monads therefore seem to provide an alternative to the extensible reasoning of section 7.6, however they never address the use of separation logic in their reasoning.

7.3 Predicate transformers

Predicate transformers (Swierstra and Baanen 2019) are another interesting development in the area of effectful verification. They provide a closely related language embedding and a extensible reasoning framework based on predicate transformers. We compare their solution to ours on three aspects: non-termination, effects and verification style. Their language is a (inductive) *free-er* monad (Kiselyov and Ishii 2015). This means they have no access to non-termination by the means of coinduction, instead they use the approach of McBride (McBride 2015) as discussed in section 7.1. This means predicate transformers have access to (one point of) general recursion, whereas Pothos has the looping combinators from the interaction trees framework. Here predicate transformers have the edge over Pothos currently, but if the environment could be opened in the future we have access to a more flexible form of general recursion.

Predicate transformers give weakest pre semantics to effects on a per effect basis, making their model more extensible than ours. They can do this more easily since they do not address a notion of separation, this makes their reasoning slightly weaker than ours and makes it easier to get the flexibility in terms of effects.

7.4 SteelCore

SteelCore on the whole is a very similar project. Their language is shallowly embedded, they allow for separation based reasoning and non-termination. SteelCore is implemented in the F-star proof assistant (Swamy, Hritcu, et al. 2016) and takes advantage of F-star specific features, that are not available in Coq. In comparison to Pothos, SteelCore has a similar model for their language based on a free monad, however they do not use the free monad structure to achieve non-termination. Instead they rely on the built-in effect `Dv` in F-star. The `Dv` effect allows for general recursion within the language of F-star. F-star disallows any use of the `Dv` effect in proofs, this keeps the logic sound and allows for easy access to turing complete programs. The way SteelCore uses the `Dv` effect is in their signature for bind which looks something like: `ctree a → (a → Dv (ctree b)) → ctree b` where `ctree` (command tree) is their type of free monad. That means that every time two values of type `ctree` are sequentially composed we have access to general recursion. The general recursion is also accessible in a much more convenient manner than the combinator based approach in Pothos. Their interpreter for their command trees also lives in the `Dv` effect, simply passing on the general recursion, until it is finally dealt with by F-star itself. SteelCore's `ctree` type is an intrinsic language, meaning that it directly carries the separation logic proofs.

Finally a big advantage of SteelCore over Pothos is their access to heterogenous heaps. SteelCore gives the programmer typed references (that may differ in type) into this heap, this feature is once again built on top of a built-in effect of F-star called `MST` (Ahman et al. 2018). The name `MST` stands for monotonic state, meaning that the heap cells have to evolve in a monotone fashion. An basic example of this would be attaching a type to the heap cell and have all values be of this type. The problem with applying this work to Pothos, is that it requires a Hoare type theory (such as F-star) or program logic while defining the language. Whereas Pothos tries to be extrinsic and leaves the inclusion of pre and post conditions until the end. Another approach could be the approach used by (Poulsen et al. 2018). They model monotone state using monads, threading information through the interpreter.

7.5 Fcsl

Fine grained concurrent separation logic (Sergey, Nanevski, and Banerjee 2015) is a development to utilize separation logic in a fine grained situations in comparison to the coarse grained concurrency done before. Coarse grained here means critical / exclusive sections (locks), whereas fine grained concurrency would be "lock-free" data structures. Algorithms that use atomic operations to co-operate without the need of locking. Their model for tackling this problem is by introducing concurrent state transfer protocols that they all *concurroids*. They sample the state space into three parts: self, joint and other. This way we know what state we can touch and what state others are dealing with for us. In Iris user defined ghost-state is able to address fine grained concurrency. FCSL can be seen as a direct competitor to Iris, but with a slightly more biased model. With no access to impredicative invariants, because they do not use step indexing. In FCSL much of the separation of heaps is taken care off by the concurroids, therefore most reasoning is done inside the regular Coq logic. This can be an advantage and a disadvantage, reasoning in the regular Coq logic is more familiar to users. But not everything can be done it, if a user does need to drop down to the level of separation logic that is easier to do in Iris.

On the language side of things FCSL has a shallowly embedded language, with a fixed-point combinator for non-termination. Their fixed-point combinator is based on domaintheory and the approximation of the denotational semantics of the program represented as sets of trees. Gaining access to non-termination in this fashion comes at a cost, namely there are proof obligations that programs are monotone in respect to this approximation. We decided we wanted to avoid such proof obligations in Pothos, hence we went with the coinductive approach. FCSL does provide a lot of built-in constructs for which the monotonicity requirement is already proven, therefore shielding the user of this proof obligation in many cases, but not all cases.

7.6 A Separation Logic for Effect Handlers

Developing solutions for the before mentioned limitation of not being extensible in the effects environment is a strain of research on its own. We briefly explore one development in this area and list the problems including it in Pothos. A Separation Logic for Effect Handlers (Vilhena and Pottier 2021) build an extensible separation logic for effects on top of Iris. They develop a notion of protocols between programs and handlers, for each effect one should create an appropriate protocol. Their definition for their weakest precondition connective is then generic in effects since it operates on these protocols. Their development is a deeply embedded language, with a direct implementation of algebraic effects that only has one-shot continuations. The deep embedding and one-shot continuations are essential for ensuring their logic has a bind rule. They have two bind rules one for ″neutral″ contexts (pieces of programs that do not include handlers) and a bind rule that works for handlers. This syntactic check would need to be translated to our shallow setting for interaction trees Finally the one-shot continuations are required for the soundness of the logic, since a multi invocation of continuations makes certain state assertions fail. The handlers given in the interaction trees framework do not give any one-shot guarantees and would also have to be adapted.

7.7 Perennial

Perennial and Goose (Chajed et al. 2019) are a tools for the verification of fault tolerant systems. Their verification is built on top of Iris and adds adds reasoning about fault tolerance. Their choice of language is completely different from ours. The programmer writes in a subset of the go programming language (Goose), which is translated into a deeply embedded Coq representation of that subset. The deep embedding is then given a formal semantics and the semantics are linked to Iris to do the verification. To increase confidence in this translation they have also developed Waddle (Gibson 2020). Waddle is a verified interpreter / test framework for Goose and Go programs. Although their interpreter does not include concurrency, it creates trust in their semantics. The interpreter has also been ported to Iris's own deeply embedded language Heaplang. Bringing Heaplang closer to Pothos in terms of executability.

It is interesting to see where we get the executional properties from and how it relates to an adequacy theorem. Pothos specifies its semantics directly in an interpreter for interaction trees and our adequacy lemma directly relates to this interpreter. Being able to do so directly is one of the benefits of our shallow embedding. Perennial on the other hand has a three step process. They have their deeply embedded language and a relationally specified semantics for that language. With the two of those they can prove their separation logic sound in respect to their semantics. They then require an extra step to prove their interpreter correct in respect to their semantics. Making this a longer process, but for a much more realistic language.

Chapter 8

Conclusion and future work

Let us first summarize what we have seen in all our chapters. We have seen Iris interact with three different monads finishing with interaction trees. We have seen our basic weakest precondition calculus for the state monad, we have experimented with non-termination and the later modality ($\triangleright$) in the delay monad setting, where we saw step indexing and introduced weakest precondition rules for the interaction trees based non-termination combinators. Finally we joined Iris and interaction trees to implement concurrency and gave the semantic basis for our logics in terms of Iris.

The first we can take away is that we now know what Iris looks like in a shallowly embedded setting. Further more we now know how to combine the features of Iris with these monads and how the combination of the two enables reasoning about these effects. Especially how to use Iris's step indexing together with coinductive types to enable non-termination in a shallow embedding.

Looking at it from the perspective of interaction trees, we know Iris based separation logic reasoning is possible for interaction trees in this restricted effect environment. Additionally we can confidently say that our logic can handle the standard Iris patterns for verifying concurrent programs, since we displayed them in our case study, giving us some confidence our logic is in fact useful. For all logics presented in this thesis we know our rules are sound, because they are derivable within Iris. We also know our definition of our weakest precondition connective are adequate for our languages due to the adequacy theorems presented in the thesis here and the fact that those theorems have been proven in our formalization.

Future work Pothos currently has two big limitations, the fact that the environment is closed and the homogenous heap. To creating a definition for a weakest precondition connective with an open effect environment there is the work of Vilhena and Pottier (2021) (as covered in section 7.6). Adapting their techniques is a substantial amount of work due to the requirement that continuations need to be one shot, the syntactic checks and the fact that their development is a deep embedding.

To address the homogenous heap, there are two approaches that could be adopted to create a heterogenous heap. First we have the work of Ahman et al. (2018) (as covered in section 7.4). They give a model for monotonic state and build references on top of this. The direct adaptation of this is difficult within Coq as it requires either a Hoare type theory (like F-star) or needs to be built on top of Iris's program logic. If we build it on top of Iris then our language always knows about its verification and has to be intrinsic. Another option would include the approach taken by Poulsen et al. (2018). They model monotone state using monads, threading information through the interpreter. This seems more directly applicable, however it requires substantial work overhauling our interpreter. Their technique also utilizes information about the name binding environment that is unavailable in our interpreter.

In conclusion there are possibilities to improve Pothos but they are non-trivial.

Bibliography

Ahman, Danel et al. (2018). "Recalling a witness: foundations and applications of monotonic state". In: *Proc. ACM Program. Lang.* 2.POPL, 65:1–65:30. DOI: `10.1145/3158153`. URL: `https://doi.org/10.1145/3158153`.

Appel, Andrew W. et al. (2007). "A very modal model of a modern, major, general type system". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, pp. 109–122. DOI: `10.1145/1190216.1190235`. URL: `https://doi.org/10.1145/1190216.1190235`.

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, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Springer, pp. 115–137. DOI: `10.1007/11804192\_6`. URL: `https://doi.org/10.1007/11804192%5C_6`.

Blom, Stefan and Marieke Huisman (2014). "The VerCors Tool for Verification of Concurrent Programs". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, pp. 127–131. DOI: `10.1007/978-3-319-06410-9\_9`. URL: `https://doi.org/10.1007/978-3-319-06410-9%5C_9`.

Calcagno, Cristiano and Dino Distefano (2011). "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru et al. Vol. 6617. Lecture Notes in Computer Science. Springer, pp. 459–465. DOI: `10.1007/978-3-642-20398-5\_33`. URL: `https://doi.org/10.1007/978-3-642-20398-5%5C_33`.

Capretta, Venanzio (2005). "General recursion via coinductive types". In: *Log. Methods Comput. Sci.* 1.2. DOI: `10.2168/LMCS-1(2:1)2005`. URL: `https://doi.org/10.2168/LMCS-1(2:1)2005`.

Chajed, Tej et al. (2019). "Verifying concurrent, crash-safe systems with Perennial". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, pp. 243–258. DOI: `10.1145/3341301.3359632`. URL: `https://doi.org/10.1145/3341301.3359632`.

Chlipala, Adam (2013). *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. ISBN: 978-0-262-02665-9. URL: `http://mitpress.mit.edu/books/certified-programming-dependent-types`.

Coq-development-team (2000a). *Co-inductive types and co-recursive functions*. URL: `https://coq.inria.fr/refman/language/core/coinductive.html` (visited on 08/06/2021).

Coq-development-team (2000b). *The Coq proof assistant reference manual.* url: https://coq.inria.fr/refman/ (visited on 08/13/2022).

Gibson, Sydney Marie (2020). "Waddle: A proven interpreter and test framework for a subset of the Go semantics". MA thesis. Massachusetts Institute Of Technology.

Hinrichsen, Jonas Kastberg, Jesper Bengtson, and Robbert Krebbers (2020). "Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic". In: *CoRR* abs/2010.15030.

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

Iris-development-team (2022). *STDPP, An extended "Standard Library" for Coq.* https://gitlab.mpi-sws.org/iris/stdpp.

Jacobs, Bart, Jan Smans, and Frank Piessens (2010). "A Quick Tour of the VeriFast Program Verifier". In: *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*. Ed. by Kazunori Ueda. Vol. 6461. Lecture Notes in Computer Science. Springer, pp. 304–311. doi: 10.1007/978-3-642-17164-2_21. url: https://doi.org/10.1007/978-3-642-17164-2%5C_21.

Jung, Ralf, Jacques-Henri Jourdan, et al. (2018). "RustBelt: securing the foundations of the rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL, 66:1–66:34.

Jung, Ralf, Robbert Krebbers, et al. (2018). "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28, e20. doi: 10.1017/S0956796818000151. url: https://doi.org/10.1017/S0956796818000151.

Kiselyov, Oleg and Hiromi Ishii (2015). "Freer monads, more extensible effects". In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. Ed. by Ben Lippmeier. ACM, pp. 94–105. doi: 10.1145/2804302.2804319. url: https://doi.org/10.1145/2804302.2804319.

Klein, Gerwin et al. (2009). "seL4: formal verification of an OS kernel". In: *SOSP*. ACM, pp. 207–220.

Leroy, Xavier (2009). "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7, pp. 107–115.

Maillard, Kenji et al. (2019). "Dijkstra monads for all". In: *PACMPL* 3.ICFP, 104:1–104:29. doi: 10.1145/3341708. url: https://doi.org/10.1145/3341708.

McBride, Conor (2015). "Turing-Completeness Totally Free". In: *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Lecture Notes in Computer Science. Springer, pp. 257–275. doi: 10.1007/978-3-319-19797-5_13. url: https://doi.org/10.1007/978-3-319-19797-5%5C_13.

Müller, Peter, Malte Schwerhoff, and Alexander J. Summers (2016). "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer, pp. 41–62. doi: 10.1007/978-3-662-49122-5_2. url: https://doi.org/10.1007/978-3-662-49122-5%5C_2.

Nakano, Hiroshi (2000). "A Modality for Recursion". In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, pp. 255–266. doi: 10.1109/LICS.2000.855774. url: https://doi.org/10.1109/LICS.2000.855774.

Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer. isbn: 3-540-43376-7. doi: 10.1007/3-540-45949-9. url: https://doi.org/10.1007/3-540-45949-9.

Norell, Ulf (2008). "Dependently Typed Programming in Agda". In: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lec-*

tures. Ed. by Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer, pp. 230–266. DOI: `10.1007/978-3-642-04652-0\_5`. URL: `https://doi.org/10.1007/978-3-642-04652-0%5C_5`.

O'Hearn, Peter W. (2004). "Resources, Concurrency and Local Reasoning". In: *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. Ed. by Philippa Gardner and Nobuko Yoshida. Vol. 3170. Lecture Notes in Computer Science. Springer, pp. 49–67. DOI: `10.1007/978-3-540-28644-8\_4`. URL: `https://doi.org/10.1007/978-3-540-28644-8%5C_4`.

— (2007). "Resources, concurrency, and local reasoning". In: *Theor. Comput. Sci.* 375.1-3, pp. 271–307. DOI: `10.1016/j.tcs.2006.12.035`. URL: `https://doi.org/10.1016/j.tcs.2006.12.035`.

O'Hearn, Peter W., John C. Reynolds, and Hongseok Yang (2001). "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, pp. 1–19. DOI: `10.1007/3-540-44802-0\_1`. URL: `https://doi.org/10.1007/3-540-44802-0%5C_1`.

Owens, Scott et al. (2016). "Functional Big-Step Semantics". In: *ESOP*. Vol. 9632. Lecture Notes in Computer Science. Springer, pp. 589–615.

Poulsen, Casper Bach et al. (2018). "Intrinsically-typed definitional interpreters for imperative languages". In: *Proc. ACM Program. Lang.* 2.POPL, 16:1–16:34. DOI: `10.1145/3158104`. URL: `https://doi.org/10.1145/3158104`.

Reynolds, John C. (2002). "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, pp. 55–74. DOI: `10.1109/LICS.2002.1029817`. URL: `https://doi.org/10.1109/LICS.2002.1029817`.

Sergey, Ilya, Aleksandar Nanevski, and Anindya Banerjee (2015). "Mechanized verification of fine-grained concurrent programs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, pp. 77–87. DOI: `10.1145/2737924.2737964`. URL: `https://doi.org/10.1145/2737924.2737964`.

Siek, Jeremy G. (2012). *Big-step, diverging or stuck?* http://siek.blogspot.com/2012/07/big-step-diverging-or-stuck.html.

Silver, Lucas and Steve Zdancewic (2021). "Dijkstra monads forever: termination-sensitive specifications for interaction trees". In: *Proc. ACM Program. Lang.* 5.POPL, pp. 1–28. DOI: `10.1145/3434307`. URL: `https://doi.org/10.1145/3434307`.

Swamy, Nikhil, Catalin Hritcu, et al. (2016). "Dependent types and multi-monadic effects in F". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, pp. 256–270. DOI: `10.1145/2837614.2837655`. URL: `https://doi.org/10.1145/2837614.2837655`.

Swamy, Nikhil, Aseem Rastogi, et al. (2020). "SteelCore: an extensible concurrent separation logic for effectful dependently typed programs". In: *Proc. ACM Program. Lang.* 4.ICFP, 121:1–121:30. DOI: `10.1145/3409003`. URL: `https://doi.org/10.1145/3409003`.

Swamy, Nikhil, Joel Weinberger, et al. (2013). "Verifying higher-order programs with the dijkstra monad". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, pp. 387–398. DOI: `10.1145/2491956.2491978`. URL: `https://doi.org/10.1145/2491956.2491978`.

Swierstra, Wouter and Tim Baanen (2019). "A predicate transformer semantics for effects (functional pearl)". In: *PACMPL* 3.ICFP, 103:1–103:26. DOI: `10.1145/3341707`. URL: `https://doi.org/10.1145/3341707`.

Vilhena, Paulo Emílio de and François Pottier (2021). "A separation logic for effect handlers". In: *Proc. ACM Program. Lang.* 5.POPL, pp. 1–28. doi: `10.1145/3434314`. url: `https://doi.org/10.1145/3434314`.

Wadler, Philip (1992). "Monads for functional programming". In: *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*. Ed. by Manfred Broy. Vol. 118. NATO ASI Series. Springer, pp. 233–264. doi: `10.1007/978-3-662-02880-3\_8`. url: `https://doi.org/10.1007/978-3-662-02880-3%5C_8`.

Wildmoser, Martin and Tobias Nipkow (2004). "Certifying Machine Code Safety: Shallow Versus Deep Embedding". In: *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*. Ed. by Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan. Vol. 3223. Lecture Notes in Computer Science. Springer, pp. 305–320. doi: `10.1007/978-3-540-30142-4\_22`. url: `https://doi.org/10.1007/978-3-540-30142-4%5C_22`.

Xia, Li-yao et al. (2020). "Interaction trees: representing recursive and impure programs in Coq". In: *Proc. ACM Program. Lang.* 4.POPL, 51:1–51:32. doi: `10.1145/3371119`. url: `https://doi.org/10.1145/3371119`.